




historia

## LOS ÚLTIMOS COLETAZOS COMERCIALES

BASIC

## ZXBASIC: UN SUEÑO HECHO REALIDAD

el aventurero

## LA HERRAMIENTA INPAWS

ensamblador

## PILA, SUBRUTINAS Y ENTRADA/SALIDA

z88dk

## SP1 Y ASM

análisis

## LA CORONA ENCANTADA

input

## ENTREVISTA A IGNACIO Y CARLOS ABRIL

opinión

## NO PASA NADA

nocilla






# Índice y *Staff*




**Redacción:**

- Miguel A. García Prada.
- Javier Vispe.
- Santiago Romero.
- Federico Álvarez.
- Pablo Suau.

**Ilustración de Portada:**

- Juanje Gómez.

**Colaboraciones en este número:**

- Josetxu Malanda.
- Albert Valls.
- Pedro J. De Celis (Pedrete).
- Carlos Sanchez.
- Boriel.

**Contacto:**

- magazinezx@gmail.com

# Editorial

Estimados lectores,

Hace más de 5 años y medio (en Julio del 2003) se publicó el primer número de **MagazineZX**. Nuestra intención con esta revista online era la de ofrecer a nuestros lectores una publicación que se acercara a la calidad de la mítica revista **Microhobby** que, para muchos de nosotros, es el mayor referente en cuanto a cómo debe de hacerse una revista.

Mientras otras publicaciones se dedicaban principalmente a llenar sus páginas con análisis de juegos y publicidad, Microhobby ofrecía (además de los ya mencionados análisis) unos excelentes artículos de programación, secciones sobre aventuras conversacionales, listados BASIC para teclear y analizar, montajes hardware y consultorios de preguntas técnicas. Muchos de nosotros aprendimos a programar con los artículos y ejemplos de esta revista.

Por eso, tratando de hacer un homenaje a la extinta Microhobby, los redactores y colaboradores de MagazineZX hemos pretendido dotar a nuestra revista de contenidos variados, sin hacer uso de la enésima revisión de clásicos del Spectrum. Nuestros artículos de entrevistas, montajes hardware, cursos de programación de Ensamblador, BASIC o Z88DK, artículos técnicos monotemáticos, o noticias de la actualidad del Spectrum tenían mucho peso en MagazineZX.

En cuanto al análisis de juegos, hemos tratado de seguir una línea que huyera de la típica selección de glorias del Spectrum ya de sobra conocidas por todos. Resultaría demasiado fácil analizar estas joyas del software y no tener la necesidad de bajar del 8 a la hora de puntuarlos. A lo largo de los diferentes números de MagazineZX hemos ido seleccionando para su evaluación (además de los nuevos desarrollos) muchos juegos desconocidos o semidesconocidos, y hemos sido realmente críticos con ellos, otorgando las notas que pensabamos que realmente se merecían, sin pararnos a pensar en quién lo desarrollaba. Y, concretamente, nos estamos refiriendo con esta última puntualización a los nuevos desarrollos, ya que existe una cierta tendencia de "no agresión" a la hora de puntuar las nuevas obras que salen a la calle.

Pero todo ciclo tiene un final. Y como se decía en aquel fatídico Editorial número 217 de Microhobby, **ésta es casi una carta de despedida. MagazineZX, vuestra Revista, la nuestra, no saldrá a la calle el próximo mes, ni tampoco el siguiente**.

Estamos seguros de que encontraréis más de un paralelismo entre aquel número 217 de Microhobby (Enero de 1992), y este número 17 (y os aseguramos que la coincidencia numérica ha sido casual) de MagazineZX. En el caso de Microhobby, el motivo del cese de su publicación era que *no querían verla degradada* por la falta de novedades. El caso de MagazineZX es similar, pero no por la falta de novedades sino por otros motivos.

Si observáis las fechas de publicación de los diferentes números, veréis que cada vez pasaba más tiempo entre una entrega y la siguiente. En nuestro caso, no es porque no tengamos noticias que dar o cosas que contar. En el área de la programación, el hardware, el análisis de nuevos desarrollos y el campo de la emulación todavía existe mucho material que tratar. Os aseguramos que más de uno de nuestros colaboradores se va a dejar en el tintero alguna entrevista o artículo interesante que ya no verá la luz dentro de MagazineZX.

No, el problema principal de MagazineZX es triple: la falta de tiempo, la falta de motivación y la falta de público.

**Falta de tiempo**, porque la gente que hacemos MagazineZX tenemos cada vez más responsabilidades y menos tiempo para dedicar a la revista. Dado que en MagazineZX ponemos la calidad por encima de todo (por lo que no aceptamos el escribir artículos menos trabajados o con menos contenido del que realmente deban tener), esto significa plazos de entrega, y por tanto de publicación, mayores. Y es por eso que el tiempo entre número y número de MagazineZX ha ido aumentando hasta llegar a un valor que, para nosotros, es inaceptable. El paso de más de un año entre la publicación del nº 16 y el nº 17 de nuestra revista nos ha demostrado que el modelo actual no puede ser mantenido.

**Falta de motivación**, porque la falta de tiempo aleja nuestras ilusiones de un mundillo en el que no podemos participar igual de activamente que lo hacíamos antes. Y algo que antes hacías con toda la ilusión del mundo y con tu tiempo libre te implica ahora un esfuerzo adicional y el robo de tiempo a otras tareas que, en este momento, te resultan más gratificantes. Básicamente, se puede decir que algunos de los integrantes de MagazineZX hemos dejado de divertirnos haciéndola. No por la revista en sí, sino porque las circunstancias de muchos de nosotros han cambiado y nos vemos desmotivados al no poder dedicarle el tiempo que quisiéramos y cumplir unos plazos mínimos de publicación.

**Falta de público**, finalmente, porque la escena (si realmente existe algo así) del Spectrum es muy reducida, y tiene un gran problema con respecto a otras revistas o publicaciones. El problema es que la audiencia de MagazineZX se divide en 2 grupos: por un lado tenemos los usuarios que no programan y no tienen interés en

las cuestiones técnicas. Para estos usuarios, la revista se reduce básicamente a la sección de noticias, alguna entrevista, y el análisis de los juegos, no encontrando utilidad al 80% de la revista. El otro grupo de usuarios, el grupo técnico, tiene una gran parte de integrantes para los cuales nuestros artículos de programación describen conocimientos que ya poseen. Esto implica que el gran trabajo que nos supone sacar un nuevo número no sólo llega a muy pocos (cuantitativamente hablando) lectores, sino que además estos lectores no le sacan provecho real al 100% de la revista.

Así, nos encontramos con falta de tiempo, falta de motivación, y muy pocos lectores (al parecer) realmente interesados en los contenidos de la revista. Recordamos dos casos concretos que pueden servir como muestra: un artículo de programación en C con un ejemplo que contenía un error (de copiado y pegado) que impedía su compilación, y un TAP del curso de ensamblador que no podía ser descargado. Que nadie repare en estos errores, así como que nadie nos envíe preguntas, dudas, sugerencias, o críticas, nos hace pensar que la cantidad de tiempo y recursos invertidos en MagazineZX no compensan como para continuar haciéndola. Porque, sinceramente, crear MagazineZX es realmente un proceso costoso de selección y composición de artículos, pero sobre todo de maquetación online y PDF.

Por eso, hemos decidido que en lugar de dejar a MagazineZX apagarse lentamente, lo mejor es darle un final digno con la publicación de un último número que pueda ser recordado por todos nuestros lectores como la mejor entrega de la revista. En él vamos a tratar de cubrir todas las noticias del mundillo del Spectrum desde nuestro último número, así como no dejar ninguna serie de artículos sin final (con este número, por ejemplo, se cerrará la totalidad del Curso de Ensamblador de Z80). Os aseguramos que hemos hecho lo posible para, superando la falta de tiempo y de motivación, dejar a vuestra disposición el número final de MagazineZX que la revista se merece.

Esperamos que disfrutéis con la lectura de este último número.

Redacción de MAGAZINE ZX



**Magazine ZX en papel**

Desde el pasado número 16, la revista aumentó los formatos en los que está disponible. Nuestro último número también puede ser adquirido en papel a través del servicio que ofrece Lulu.com. Hazte con el tuyo. ¡Dentro de unos años se cotizará a precio de oro en ebay!

**25 años de QL**

Cifra mágica que Salvador Merino, Badaman y la gente de "SinclairQL - Recursos en castellano" celebra con una vasta recopilación de documentación, manuales y revistas sobre el ordenador que seguro aprovecharán los usuarios del sistema.




*QL 25 aniversario*

**Retroacción en 2008**

La asociación Retroacción lleva ya varios años celebrando dos de los eventos más importantes sobre retroinformática de nuestro país: Retroeuskal y Retromañía. Aunque el pasado año hubo una especial atención a las videoconsolas, con actividades y exposiciones mostrando las marcas y modelos más significativos de la historia, no dejó de lado el Spectrum. Programadores de la talla de Alberto González (New Frontier) o Ignacio Abril (Dinamic), y expertos del hardware como José Leandro Novellón y Miguel Ángel Rodríguez (mcleod_ideafix) participaron en diversas charlas dedicadas en parte o totalmente a los ordenadores de Sinclair. Disponéis del audio, y del resto de actividades, en la web de la asociación.




*Retromañía '08*

**La actualidad del soft**

En 2008 la producción de videojuegos ha mantenido la continuidad a través de diversos grupos y programadores. Hagamos un repaso general.

*Como en los viejos tiempos: La Corona Encantada*

Con caja de lujo y carátula de Azpiri, así se presenta Karoshi en su nuevo desarrollo para Spectrum (y MSX): La Corona Encantada, del que damos cuenta en la sección Al Descubierto.

*La despedida y el retorno del Doctor Van Halen*

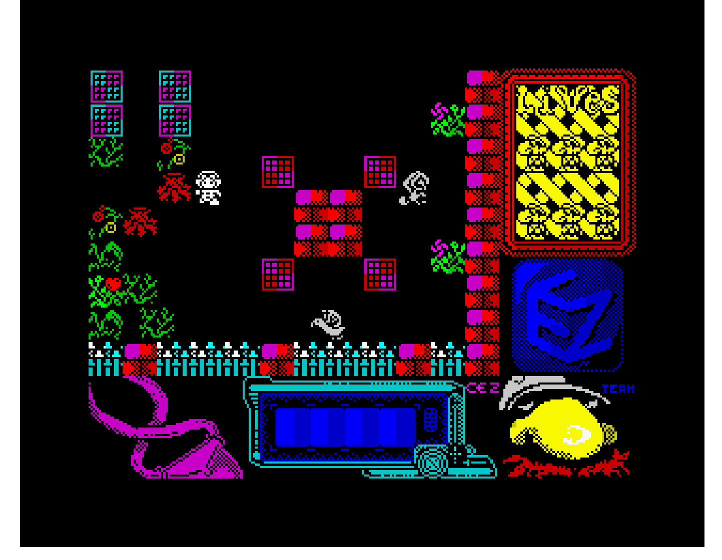
Josep Coletas dejó la actividad en el mundo de las aventuras conversacionales con dos entregas de su personaje más popular. Una vez más, el CAAD fue el encargado de alojar a "Las calles del miedo" y "Las Ruedas de Ezequiel". Si bien, para sorpresa y alegría de sus seguidores, acaba de lanzar una nueva entrega: "Los Extraordinarios Casos del Dr. Van Halen Volumen 2 Relato III: El Hijo del Crepusculo."

*Cronosoft*

La editora británica de Simon Ullyatt, no ha pasado por sus mejores momentos a final de año. Las escasas ventas, unidas a imprevistos circunstanciales, han puesto en riesgo la continuidad del proyecto. Bob Smith finalizó su trilogía del granjero Jack basada en arcades clásicos, y se lanzó a por la baja resolución con “SplATTR”. Jonathan Cauldwell ha traído, como no, un nuevo Egghead, y utilidades para crear arcades y shoot'em ups sencillos. A estos nombres hay que añadir el de Kevin Thacker con su “Sudoku”.

*Otra despedida: CEZGS*

En 2008 la actividad del grupo se ha visto como de costumbre reflejada en la salida de varios juegos en diferentes plataformas: “JINJ”, “Mariano the dragon”, “Ilogicall”,… Además asistimos a la apertura de su tienda online. Sin embargo, en Enero Karnevi anunció por sorpresa la disolución del colectivo. Varios de sus miembros ya han comentado la intención de continuar con nuevos proyectos. Mientras llega el adiós definitivo, Metalbrain ha publicado dentro del sello su primer juego: “I need speed”.



*JINJ*

*Otro Sudoku, pero para Sinclair QL: "OSUSQ"*

Afx desde el foro de QL de speccy.org se encargó de portar este juego numérico que no necesita presentación. Un sudoku con capacidad para ofrecer al usuario difíciles partidas y de resolver sudokus automáticamente.

*Nonocross*

Sin hacer ruido, Alberto Otero Mato bajo el nombre de Digital Brains presentó "Nonocross", basado en los nonogramas que han hecho furor en Nintendo DS. Se trata del primer trabajo de Alberto del que tenemos noticia.

*Juegos rusos*

Desde el este hemos contemplado algunas nuevas producciones, como "Nipik 2" de Triumph game labs o el enigmático "Vera", del que esperamos algún día poder ver la traducción al inglés anunciada.



*Vera*

**Emuladores**

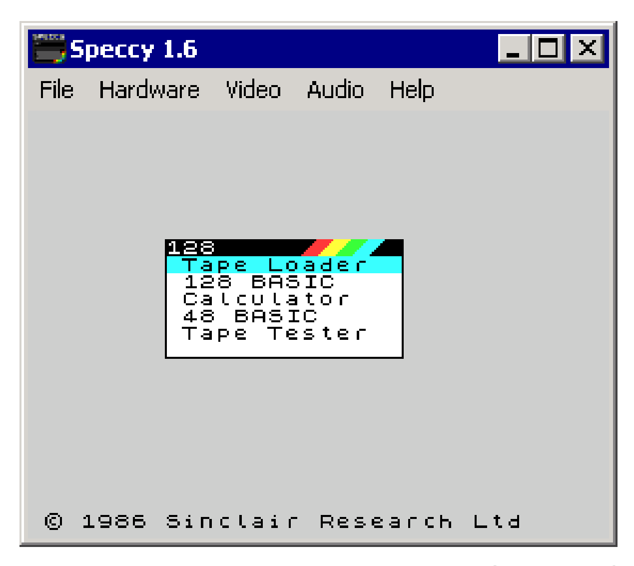
Otra sección que no ha faltado en los últimos números de Magazine ZX. El Spectrum sigue extendiendo su leyenda por plataformas y sistemas operativos que nunca soñó.

Los usuarios de Linux han de estar pendientes de las varias opciones de que disponen. La más popular es Fuse, el emulador de Phillip Kendall, que ya ha llegado a la versión 0.10.0.2. Si originariamente su objetivo primario eran los sistemas Unix, a día de hoy puede ser usado en Windows, Mac OS X, Amiga OS, MorphOS, consolas como Wii, PSP o los terminales N800/810 de Nokia. Algunas de las novedades en las últimas actualizaciones han sido la emulación del +D, la mejora en el chip upd765 o el controlador de disco del +3.

Un pequeño inciso para referirnos a otro emulador que también funciona en ordenadores Apple al margen de Fuse. ZXSP ya no sólo ofrece acceso al Spectrum, sino también a máquinas como el ZX80, ZX81 y Jupiter Ace.

Volviendo al sistema que nos concierne, hablemos de FBZX, que llegó a la versión 2.0. Programado por Sergio Costas, ha conseguido tener el 100% de su código libre. Una opción más para este Sistema Operativo es Higgins, procedente de tierras rusas, y, que, según dicen sus autores, quieren conseguir dar la sensación que provoca un Spectrum real.

Otra de las opciones que se apuntan a linux es Speccy. A partir de la 1.6 Marat Fayzullin lo ha portado a este sistema. Las mejoras se centraron en añadir los clones Pentagon y Didaktik, y la emulación de modos gráficos extra y el turbo(7 MHz) de los modelos rusos.




*Speccy 1.6*

Obviamente, en los sistemas operativos de Microsoft también ha habido novedades. Spectaculator de momento se queda en la versión 7.0.1. Jonathan Needle ha aportado una larga lista de mejoras, afectando a temas como las opciones de pantalla, la inclusión de un teclado virtual, emulación del plus D,...

Eightyone llegó hasta la revisión 1.0. Sin embargo, los únicos cambios que reportó fueron la compatibilidad con Windows Vista. Lo que empezó siendo un emulador de ZX81, ahora nos permite incluso emular el +3e y manejar las Compact flash preparadas para interfaces IDE.

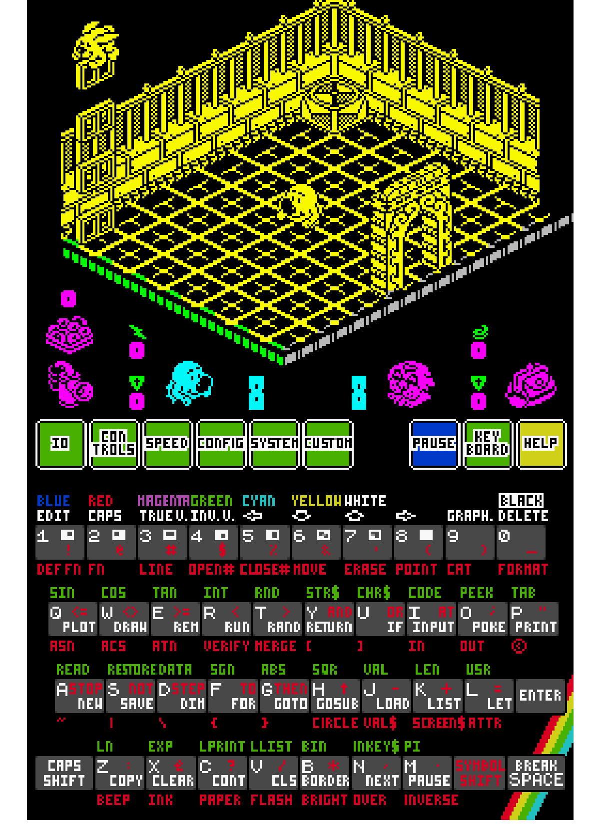
Hablando de otros habituales, debemos citar a Specemu 2.7. Mark Woodmass ha seguido invirtiendo horas en llevar la contraria a la definición que asigna a su emulador: uno de los peores del mundo. Leniad ha ampliado las máquinas emuladas en DSP 0.9ß1 (como por ejemplo el cpc) sin dejar de mejorar el resto de arcades y ordenadores ya incluidos.

Por último, hay que hablar de Spud. Richard Chandler se estrenó en el mundo de los emuladores empezando por el modelo de 48k. En sucesivas versiones lo ha dotado de más modelos, así como de un desarrollo continuo de las diferentes opciones que presta su programa. ZXSpectr, destinado al clásico MS-DOS también recibió pequeñas actualizaciones, como cambios en algunos menús, las funciones de emulación de cinta y con grabación de audio y vídeo en nuevos formatos.

Si nos referimos a las videoconsolas, en sobremesa, al margen del Fuse para Wii sólo ha habido movimiento en Dreamcast, con la salida de una nueva beta de ZX4all.

En el terreno de las portátiles, Metalbrain ha contado con colaboraciones en el desarrollo de GP2Xpectrum, como las de Seleuco o headoverheels. Gracias a ellas, en el emulador se han obtenido muchísimas mejoras de sonido y el soporte de archivos tzx.

Para Nintendo DS, Patrik Rak sigue invirtiendo horas en ZXDS. En la versión 8, dotó al emulador del sistema TR-DOS, mejoras para el formato TZX, diferentes modos de vídeo y captura de pantalla, así como implementación de nuevas funcionalidades para el sonido del chip AY. También se incluye reproducción de archivos RZX, soporte para archivos comprimidos en zip, soporte del formato POK para POKES, posibilidad de apagado de la pantalla inferior y "sleep mode". Casi nada.




*ZXDS*

Para terminar el repaso de los Sinclair virtuales, haremos referencia a JSSpeccy, criatura surgida de los dedos de Matt Wescott y que funciona en Javascript.

**ZX Spectrum in the 21st Century?**

Richard Tarjan lleva ya tiempo recopilando todo lo relacionado con el nuevo hardware y software producido para Spectrum desde 1993. En sus páginas encontraremos datos sobre prototipos, clones, periféricos,… El autor recibe de buena gana comentarios que le ayuden a completar la información disponible.

**Z80 chip MUSIC SITE**

Web dedicada al sonido creado en los Spectrum de 48 o menos K a través del Z80 y la ULA. yerzmyey recoge en formato mp3 multitud de composiciones musicales, con piezas de clásicos como Tim Follin o David Whittaker, y por supuesto, autores contemporáneos.

**ZXTUNES.COM**

Sin dejar el asunto musical, también hay que citar la creación de un archivo dedicado a recopilar los diversos trabajos realizados con las capacidades sonoras del Spectrum. Su responsable, Newart, ya tiene registrados en la base de datos casi 1000 autores y más de 22000 composiciones.

**Proyecto BASIC**

Ignacio Prini inauguró en 2008 una iniciativa que pretende preservar todo el material disponible en cuestión de listados de programas de revistas y libros. Contando con la ingente recopilación ya realizada por el proyecto "K y Enter" del Elfo Oscuro quiere ser otro referente al estilo de WOS y SPA2, pero expresamente dedicado a este ámbito.

**El tebeo informático**

El diario "El País", en su suplemento "Pequeño País", publicó más de un centenar de artículos dedicados al software de ordenadores de 8 bits. Gracias al trabajo de Fede Álvarez, están disponibles en una sección del Trastero del Spectrum.


Si lees...

'Exolon'

Truco Amstrad

De propina

*El Tebeo Informático*

**Charla de Gominolas**

Gracias a la colaboración de Horace, RetroMadrid pudo contar con una charla realizada por César Astudillo. Más conocido como Gominolas, se encargó de poner la banda sonora a muchos de los juegos de Topo soft.

**The Your Sinclair Rock'n'Roll Years. the documentary: 1989**

Nick Humphries llegó a 1989 dentro de la serie de vídeos que dedica en forma de documental a la vida comercial del Spectrum.

**SPA2 actualiza**

Aunque no como hubiéramos deseado. La gran cantidad de material recopilado en los últimos años no podía quedar a la espera de la nueva web del proyecto. Así pues, con motivo del cumpleaños de nuestra página hermana, “The World of Spectrum”, se pone a disposición pública el trabajo de más de 3 años.

**Concursos bytemaniacos**

Durante 2008 Radastan siguió convocando los concursos habituales, aunque se ha notado un descenso en la participación e incluso alguna categoría como la de Interface II ha quedado desierta. En la convocatoria de conversacionales resultó ganador “Despertar, la cueva”, mientras que en basic se llevaron el gato al agua “Relatos de Ahion” y “Asucar!”.

**SpeccyTour 2008**

La última edición del speccy tour trajo consigo un listado cerrado de juegos, del cual se debía elegir la cantidad que formaran parte de la competición. De entre los 10 participantes salió Alessandro Grussu como ganador.

**The ZX Basic Compiler Project**

Boriel ha ido desarrollando durante el pasado año un compilador de un dialecto basic que permite crear código para un Spectrum. Permite desarrollar fácilmente programas para Spectrum con un aprendizaje mínimo respecto a lo que es Sinclair BASIC. Los programas compilados con ZXB Compiler se ejecutan a una velocidad muy superior a las capacidades de interpretación del Sinclair Basic. Por si fuera poco, Boriel ha extendido el lenguaje proporcionando todo tipo de comodidades para el programador: funciones, ensamblador inline, bibliotecas externas de todo tipo, etc.

Los foros de speccy.org han servido para ir comentando todo lo acontecido durante su implementación. De hecho, allí es posible contactar fácilmente con su autor a través del subforo de Desarrollo.

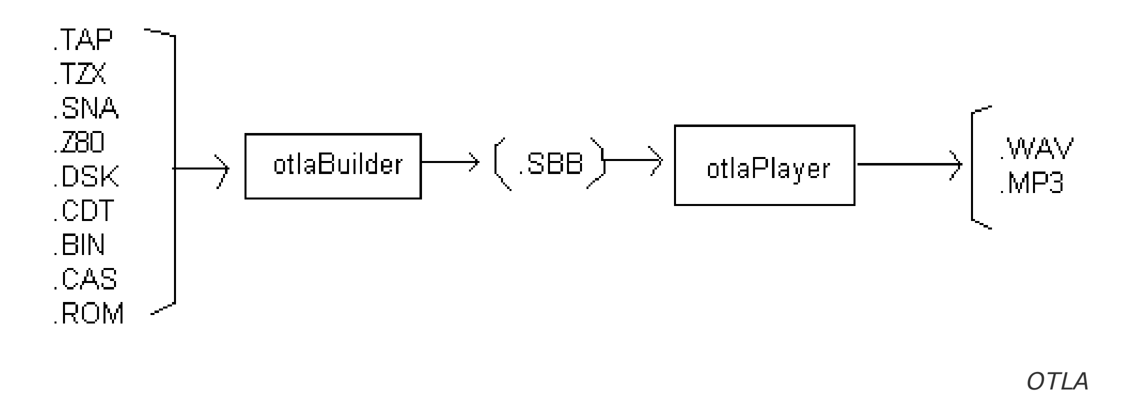
**Tommygun**

Kiwi ha llegado a la versión 0.9.37 de su entorno de desarrollo para juegos en sistemas de 8 bits. Sucesivas revisiones han solucionado bugs y añadido mejoras, como el soporte para la salida de datos en el mismo formato que SevenuP, novedades en el editor de mapas y en el compilador.

**Proyecto OTLA**

Otla es una herramienta que sirve para cargar programas en ordenadores ZX Spectrum, Amstrad CPC y MSX

por la entrada de cassette, pero a una velocidad mucho mayor al aprovechar la tecnología de los sistemas de audio digital. En él se ha integrado K7zx, la utilidad desarrollada anteriormente por decicoder para la creación de ultracargas.




*OTLA*

**Basin**

Las nuevas versiones del editor de Basic de Dunny que han aparecido en el año anterior han solucionado multitud de errores, además de la inclusión de un editor de pantallas y MIDI.

**Una de compiladores**

En Marzo apareció la versión 1.8 de z88dk, el compilador cruzado multisistema de Alvin Albrecht. Un mes después le seguía SjASMPlus con la "RC 7". En el caso de Z88DK, cabe destacar la corrección de bastantes bugs y la finalización de la integración iniciada en la versión 1.7 de la librería SPLIB dentro del entorno del compilador. La nueva SPLIB3, rebautizada como SP1, forma ahora parte del compilador y puede ser utilizada cualquiera de sus partes sin necesidad de utilizar la librería completa, puesto que se ha dividido en módulos aislados.

**Retro-X**

Leszek Chmielewski (LCD), autor del conversor gráfico BMP2SCR lleva tiempo desarrollando a su sucesor. La nueva herramienta cuenta con editor gráfico, visor y conversor. Destaca la gran cantidad de sistemas para los que está preparado: Spectrum, SAM Coupe, Sprinter 2000, Commodore 64, Amstrad CPC, MSX, Atari XL/XE…

**Publicaciones**

En cuestiones de fanzines, haremos notar la presencia de 3 nuevos números de Bytemaniacos, y el cierre temporal de ZX Spectrum Files después del número 12. Parece que su autor ha vuelto a desdecirse (tras justificar su retirada) y ha lanzado un nuevo número. Habrá que ver hasta cuándo le durará el combustible a esta nueva etapa.

No todo van a ser despedidas, y desde el CAAD ha comenzado su andadura SPAC. Hasta el momento, tres números sobre el mundo de las conversacionales.



*SPAC*

Por otro lado, en el ámbito internacional, Paul Johns ha gestionado el proyecto "Spectrum games bible". Una serie de libros donde se analizan cronológicamente juegos de Spectrum. Ya están listas las cuatro primeras entregas en lulu.com.

**ZXUSB Interface**

Sami Vehmaa ha continuado con sus proyectos de hardware. Con ZXUSB consiguió conectar dispositivos de almacenamiento a través de este estándar. El siguiente paso fue fusionarlo con ZXCF+2 y convertirse en el ZXMatrix, interface para conexión de tarjetas Compact flash y dispositivos USB. De momento, Garry Lancaster ya le ha dado soporte en ResiDOS.

**ResiDOS y +3e actualizados**

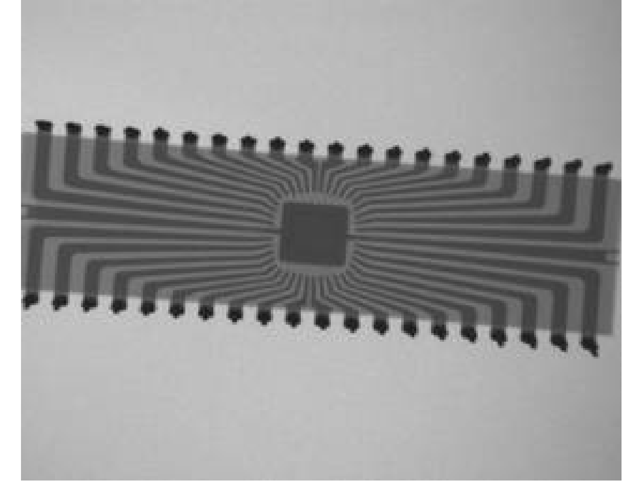
Y hablando de Garry, sus dos proyectos han seguido en desarrollo. ResiDOS, ya en su versión 2.07 ahora se puede actualizar desde la tarjeta o el disco duro, y como se ha comentado, da soporte al ZXCF+2, ZXUSB y ZX-matrix. Las roms del +3e recibieron la correspondiente revisión de fallos y también se adaptaron al interface MB02+ y MB02+IDE.

**Demfir d0.8a**

Los diferentes firmwares que puede usar el DivIDE llevan bastante tiempo parados. El único que se ha actualizado es Demfir, que por primera vez permite guardar datos en archivos TAP.

**Repuesto para ULA**

Chris Smith, el autor del proyecto Harlequin, ha conseguido recrear una ULA para 48K por medio de CPLD, la cual permitirá reemplazar el integrado original. Además ha accedido al diseño interior de la ULA. Los resultados de sus investigaciones serán recogidos en un libro denominado "The Sinclair ZX Spectrum ULA", que se pondrá a la venta a través de lulu.com en los próximos meses. Una información sin duda muy interesante que podría evitar el tener que almacenar repuestos para nuestras ULAs así como el acoplamiento de disipadores de calor en las ULAs de Spectrums en funcionamiento para alargar su vida útil.



*ULA*

**Adaptador de teclado PS/2 externo para Spectrum**

Ben Versteeg, un aficionado holandés del Spectrum, ha realizado una pequeña tirada del adaptador de teclado PS/2 externo para Spectrum que diseñó mcleod_ideafix. El interface permite seguir conectando más periféricos en el slot de expansión, y ya ha comprobado su compatibilidad con interfaces como DivIDE, MB02, Kempston mouse turbo… No sabemos si quedan unidades sobrantes, pero siempre se le puede preguntar a través de su blog.

**Interface de video compatible MSX para ZX Spectrum**

Otro de los múltiples proyectos realizados por el sevillano mcleod_ideafix es un añadido que permite manejar el chip de vídeo del MSX en nuestros Spectrum. Este interface permitiría a los programas que lo utilicen el aprovechar las capacidades de "procesado hardware" del chip del MSX1 para trabajar con sprites o tiles de una forma que el Spectrum no puede, liberando además al procesador Z80 de ello.

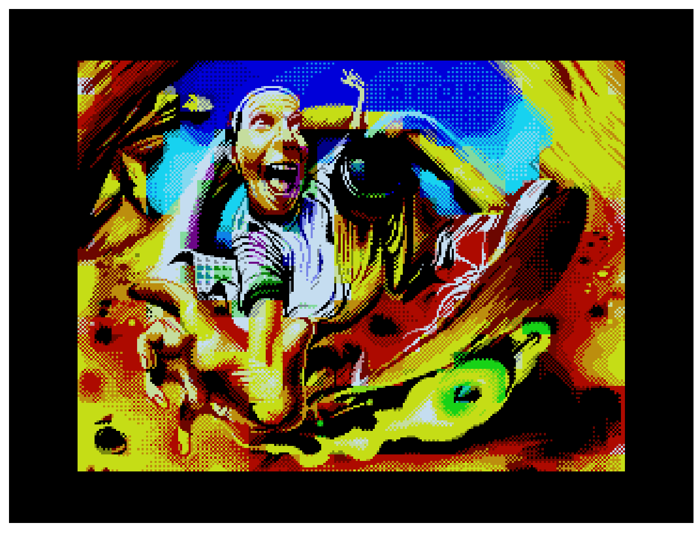
**Spectranet**

El proyecto de conexión Ethernet para Spectrum va tomando forma en las manos de Winston. Ya hay varias placas prototipo funcionando, y se ha probado con éxito alguna aplicación. Veremos que nos depara a lo largo de este año.

**Demoscene**

Para cerrar la sección de panorama, nos detendremos en el repaso a la actualidad de la demoscene. Una de las noticias más destacadas fue la victoria de Gasman en la categoría "extreme" de música en la Assembly 2008, la party más importante a nivel mundial en este campo. Dentro del panorama retro se han celebrado las tradicionales Ascii 2008, Chaos Constructions, DiHALT, Forever o las nominalmente provocativas International Vodka party y raww.orgy.

El panorama nacional, bastante escaso en cuanto a arte vintagenario, ha visto como Retroeuskal celebraba su primera compo gráfica. Con abundante presencia de producciones para spectrum, quien se llevó el primer puesto fue Almighty God con su creación para Commodore 64 "Show the ruler (Who rules the universe)".



*Shepherd's Truecolor Swings (Riskej)*

**inPAWS**

El 14 de Febrero del 2009, por sorpresa, apareció una nueva herramienta orientada a facilitar enormemente el ciclo de desarrollo de aventuras conversacionales basadas en la herramienta PAW (Professional Adventure Writer) de Gilsoft. Nos referimos a inpaws, que nos permite generar una base de datos de PAWS a través de la

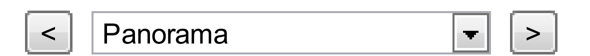
compilación de un fichero de texto con una sintaxis determinada.

Podéis obtener más información de inPAWS en este mismo número de MagazineZX, con el artículo dedicado a esta herramienta, o en la nueva Web de inpaws, alojada en Speccy.org.

**LINKS**

- CAAD: http://www.caad.es/
- Web de Bob Smith: http://www.bobs-stuff.co.uk/
- Cronosoft: http://www.cronosoft.co.uk/
- FUSE (1): http://fuse-emulator.sourceforge.net/
- FUSE (2): http://sourceforge.net/projects/fuse-for-macosx/
- FUSE (3): http://psp.akop.org/fuse
- SpecEmu: http://homepage.ntlworld.com/mark.woodmass/specemu.html
- ZXSP: http://k1.dyndns.org/Develop/projects/zxsp-osx/distributions/
- GP2Xpectrum: http://www.speccy.org/metalbrain/GP2Xpectrum1.7.2.zip
- DSP: http://www.leniad.cjb.net/dsp/index_es.htm
- Speccy: http://fms.komkon.org/Speccy/
- Unreal Speccy: http://alonecoder.narod.ru/zx/index.html
- ZX4All: http://chui.dcemu.co.uk/zx4all.html
- Eightyone: http://www.chuntey.com/eightyone/
- Spectaculator: http://www.spectaculator.com/
- ZXDS: http://zxds.raxoft.cz/
- ZX Spectr: http://sourceforge.net/projects/zxspectr/
- JSSpeccy: http://matt.west.co.tt/spectrum/jsspeccy/
- FBZX: http://www.rastersoft.com/programas/fbzxesp.html
- Higgins: http://jhiggins.narod.ru/
- Webs de hardware y software post-1993: http://tarjan.uw.hu/zxclones_en.htm http://tarjan.uw.hu/zx_gamez_after_93_en.htm
- Z80 chip MUSIC SITE: http://z80.i-demo.pl/
- ZXTUNES.COM: http://zxtunes.com/
- Proyecto BASIC: http://microhobby.speccy.cz/zxsf/listados/index.htm
- El Tebeo Informático: http://www.speccy.org/trastero/cosas/Revi/El_Tebeo_Informatico/Tebeo.html
- The Your Sinclair R'N'R Years: http://www.ysrnry.co.uk/tvprog/
- SPA2: http://spa2.speccy.org/
- Speccy Tour: http://speccytour.blogspot.com/
- The ZX Basic Compiler Project Page: http://zxbcompiler.speccy.org/
- Tommygun: http://www.users.on.net/~tonyt73/TommyGun/
- Otla: http://code.google.com/p/otla/
- Basin: ftp://ftp.worldofspectrum.org/pub/sinclair/emulators/pc/windows/BASin_r14c.exe
- z88dk: http://www.z88dk.org/
- SjASMPlus: http://sjasmplus.sourceforge.net/
- Bytemaniacos: http://www.bytemaniacos.com/
- ZX Spectrum Files: http://microhobby.speccy.cz/zxsf/pagina_1.htm
- SPAC: http://spac.caad.es/
- Spectrum Games Bible: http://www.spectrumgamesbible.co.uk/
- Web de Sami Vehmaa: http://user.tninet.se/~vjz762w/
- ResiDOS: http://www.worldofspectrum.org/residos/
- ZXSpectrum +3e: http://www.worldofspectrum.org/zxplus3e/
- Demfir: http://demfir.sourceforge.net/
- Web de Chris Smith (ULA): http://www.zxdesign.info/
- Interface teclado (Ben Versteeg): http://spac.caad.es/
- Interface de vídeo compatible MSX: http://www.zxprojects.com
- Spectranet: http://spectrum.alioth.net/doc/
- Inpaws: http://inpaws.speccy.org

Redacción de MAGAZINE ZX

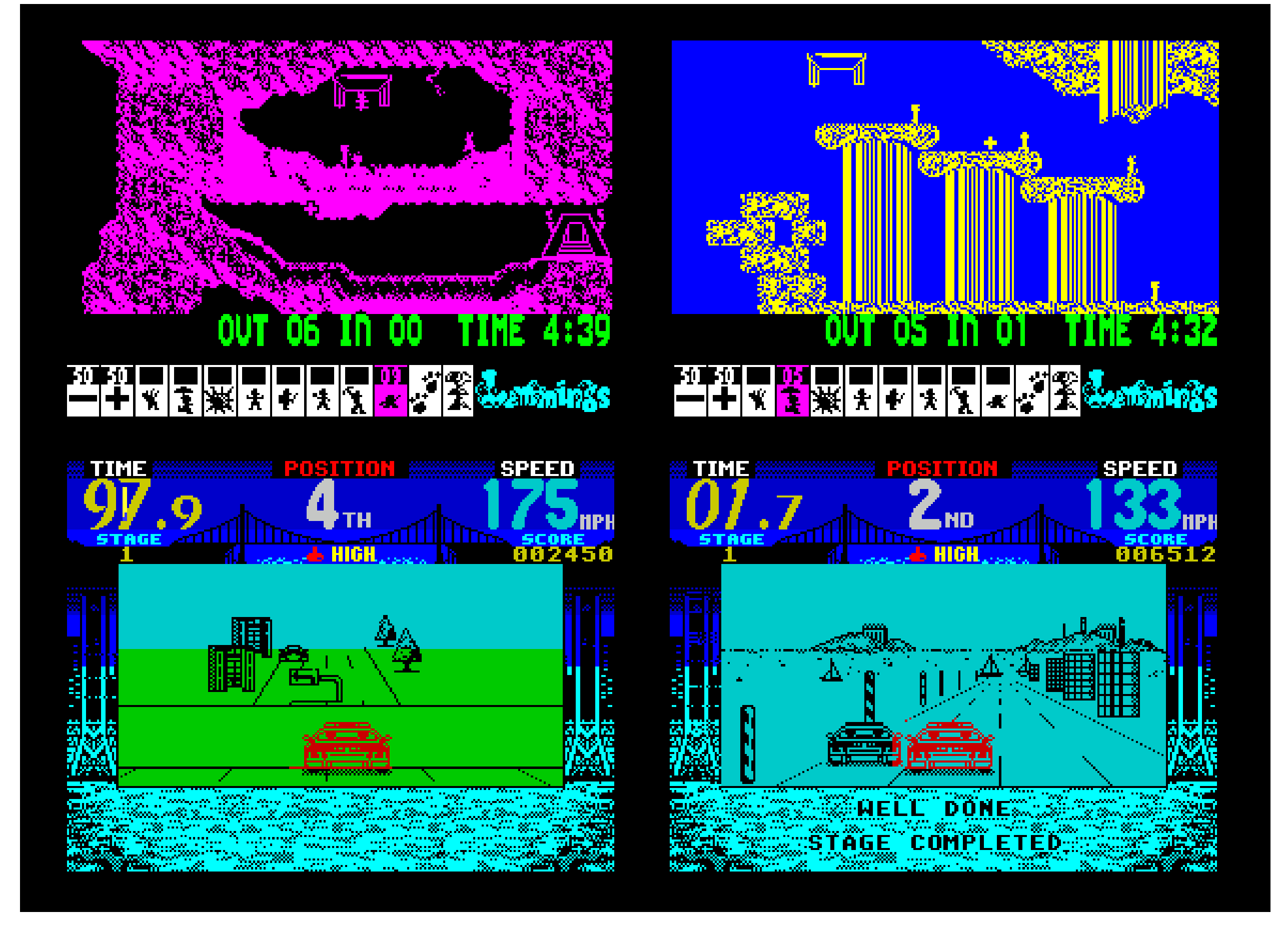


# Los últimos coletazos comerciales del Spectrum

Siendo éste como es un número de despedida, es imposible no recordar otra despedida como fue la que tuvimos que sufrir hace ya muchos años: el último número de la revista MicroHobby. Nuestra afición por el Spectrum, y por extensión, por las máquinas de 8 bits en general está fuertemente fundamentada en una nostalgia que empezamos a vivir incluso ya en esos lejanos primeros meses del año 92. Es por ello que el tema del último número de la revista MicroHobby ha sido tratado hasta la saciedad y por tanto se hace innecesario redundar en ello aquí. Sin embargo, hay algo interesante sobre este asunto: todo el mundo se llevó una sorpresa al leer la editorial de la **MicroHobby 217**, nadie se lo esperaba. Y sin embargo, si superamos esta misma nostalgia que hace embellecer todo lo que no recordamos con exactitud, puede que seamos capaces de reconocer que, aun siendo una sorpresa total, el final de la era comercial del Spectrum estuvo marcada por un largo proceso de decadencia. No sé qué pensará el lector sobre el asunto, pero mi opinión personal sobre juegos de finales del 91 y principios del 92 es que en su inmensa mayoría eran bastante malos. Sumidos en un infructuoso afán por llevar nuestra máquina más allá de sus posibilidades reales y de simular lo que ya podíamos disfrutar en las recreativas o en los nuevos ordenadores personales domésticos, éstos últimos juegos compartían muchas características en común que los hacían poco atractivos o directamente injugables: un incremento masivo en la duración y la cantidad de las cargas, grandes sprites monocromos cuyo color era simplemente el color de fondo de la zona de la pantalla por la que pasaban, pantallas sobrecargadas, movimiento lento, etc. ¿Quién no recuerda Gauntlet 3, Indiana Jones y la última cruzada, Pit Fighter, y otros engendros de este tipo?



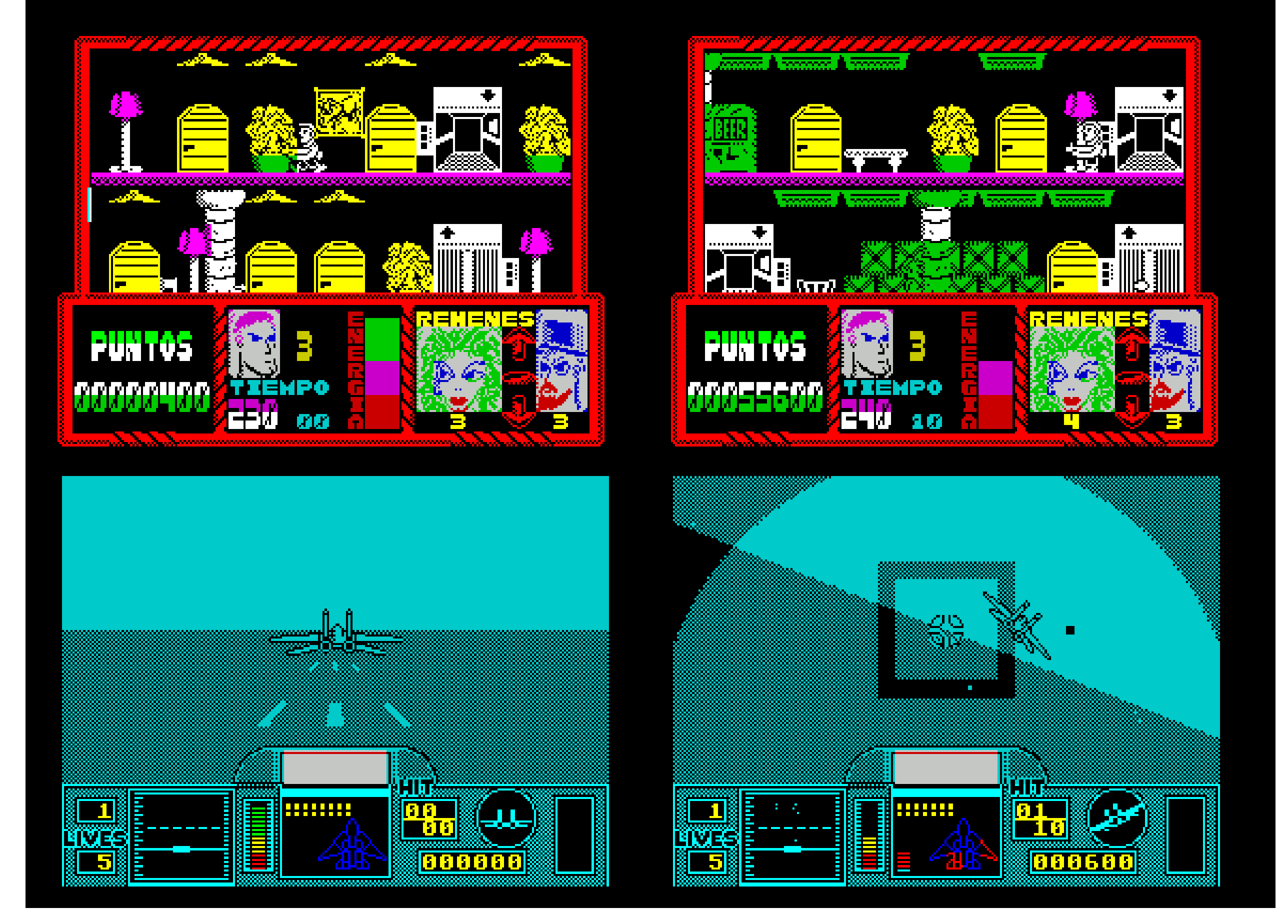

Hagamos, por ejemplo, un ligero repaso a las últimas novedades presentes en la anteriormente mencionada MicroHobby 217. En primer lugar tenemos el juego protagonista de la portada, **Lemmings**. Vistas las capturas, es factible pensar que la conversión a Spectrum era bastante buena. Y es cierto que los gráficos son muy agradables, y los niveles son los mismos que podíamos encontrar en cualquier otra versión. Pero si miramos más allá del aspecto externo, empezamos a ver problemas. En primer lugar, cada nivel se cargaba por separado, lo que desde luego era algo muy incómodo si se jugaba en cinta. Por otra parte, los controles eran desastrosos. Sin la sensibilidad propia de un ratón, muchas veces nos encontrábamos en la situación de que el puntero se movía alrededor de un maldito lemming sin que consiguiéramos atinar. Los niveles estaban bien reproducidos, pero la baja resolución impedía que cupieran completamente en el área de visualización, por lo que era necesario acudir al scroll (lo cual unido a los malos controles era también muy incómodo). Por último, el área de juego era totalmente monocroma, por lo que en algunas ocasiones era muy difícil distinguir los lemmings del entorno.




*Lemmings y Cisco Heat*

Podemos continuar hablando de **Cisco Heat**, un juego de carreras protagonizado por un coche de policía donde la velocidad se intenta simular con una animación muy brusca de la carretera y de los elementos exteriores a ésta, que lo único que consiguen es despistar y que choquemos muchas veces. Música 128K insulsa y repetitiva, y unos efectos de sonido limitados a dos únicos ruidos. Era imposible escuchar el sonido del motor de tu propio coche. Otra vez gráficos totalmente monocromos, a excepción del coche del protagonista que variaba su color con respecto al resto de elementos de la pantalla con el fin de que la confusión al menos disminuyera un poco. También se trata de un juego multicarga, aunque afortunadamente en esta ocasión el número de fases era bajo y éstas cargaban con relativa rapidez.

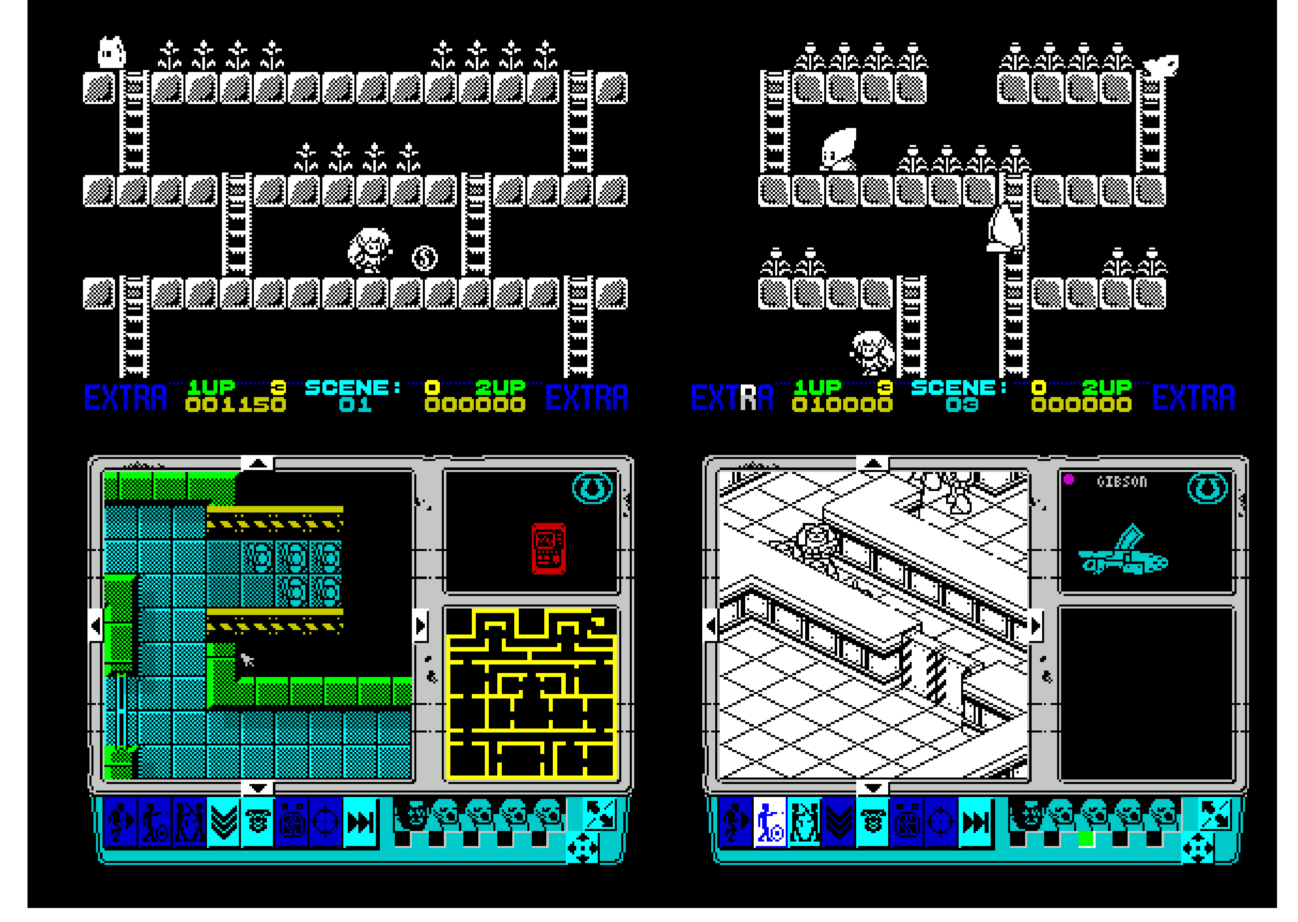
Zigurat presentaba **Piso Zero**, un programa que no llegaba a la altura de sus éxitos anteriores. Se trataba de un juego de rescate de prisioneros que guardaba muchas similitudes con el clásico Elevator Action. En esta ocasión debíamos ir viajando arriba y abajo por un edificio mientras eliminábamos a los malvados que aparecían por las puertas, intentábamos esquivar sus disparos (lo cual se traducía en que prácticamente nos pasábamos toda la partida agachados) y tratábamos de distinguir qué era y qué no era un rehén (algo que normalmente descubríamos tras disparar y ver como la cuenta de rehenes descendía). El rápido y caótico movimiento de los secuestradores hacía de Piso Zero bastante poco jugable. ¿Y qué decir de la conversión del **G-Loc** de Sega? Otro juego con sprites monócromos y monótonos, efectos sonoros paupérrimos y una jugabilidad muy dudosa, que sin duda no llega a la altura del clásico After Burner que intenta superar.




*Piso Zero y G-Loc*

De todas formas, no sería justo dar una impresión negativa sobre todos los juegos que aparecieron en dicho número. Tomemos el ejemplo de **Rodland**, un plataformas de los de toda la vida, al estilo Bubble Bobble. Era un producto más que aceptable, aunque sus agradables sprites y trabajadas animaciones no conseguían disimular del todo sus carencias. ¿Por qué juegos anteriores del mismo estilo como Rainbow Islands sí tenían una agradable melodía 128K durante el juego e iban más allá del blanco y negro y Rodland, posterior, no? Sin duda un misterio difícil de resolver.

Por su parte, **Space Crusade**, que aunque no analizado en profundidad sí que es reseñado brevemente, sorprende con su calidad. Sigue la misma mecánica que su precursor en los juegos de tablero, Hero Quest. La animación inicial del título (increíble el filtro de partículas y la velocidad a la que funciona) da pie a pensar que detrás de Space Crusade hay buenos programadores, aunque el tipo de programa no permita hacer alarde de mucha calidad técnica. Quizá el secreto de su atractivo fue que sus desarrolladores fueron realistas con respecto a las capacidades de la máquina. Curiosa también la pequeña noticia que se podía leer en ese último número de la desaparecida MicroHobby y que relataba la publicación de un emulador de Spectrum para PC. Hay quien ni se esperó a la defunción oficial de la máquina para comenzar a añorarla.




*Rodland y Space Crusade*

Como se ha indicado anteriormente, la tendencia seguida por los juegos comentados en el último ejemplar de la revista no era más que la continuar con la larga espiral de despropósitos que tuvimos que sufrir a lo largo de los últimos meses del año 91. Entre ellos cabe destacar **Pit Fighter**, con una carga interminable y unos sprites desproporcionados en cuanto a dimensiones, brusquedad y lentitud. Recordado como uno de los peores juegos de lucha de la historia.

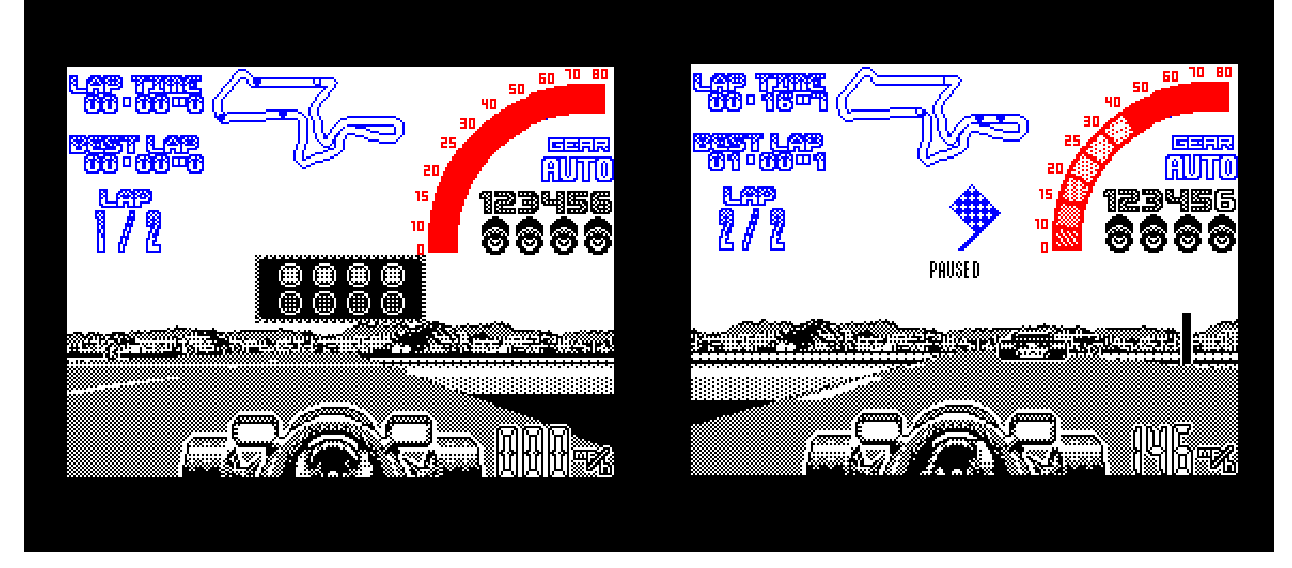
Algo mejor fue **WWF Wrestlemania**, que a pesar de ser demasiado simple y monótono tuvo buenas ventas debido sin duda a la licencia. **Double Dragon III** no era más que una burla de su primera parte. Con **Gauntlet III** me llevé una gran decepción en mi adolescencia.

**Bart Simpson and the Space Mutants** no era realmente un juego tan malo, e incluso se salía de la línea general en cuanto a características técnicas. Qué lástima que sufriera de un control insufrible del personaje protagonista, que parecía patinar por el escenario, y de una gran dificultad para atinar con el spray en los objetivos. Y desde luego el Spectrum no estaba preparado para emular el maravilloso **Alien Storm** de Sega, al menos no de la forma en la que lo fue hecho.

Pero, aunque a algunos les pudiera parecer lo contrario, el final de la revista MicroHobby no marcó también el final de la vida comercial del Spectrum. Allá en tierras inglesas, dos de las revistas de mayor importancia relacionadas con el Spectrum, **Crash Magazine** y **Sinclair User**, se unen en una publicación conjunta en Marzo del año 92, que seguirá saliendo en los kioskos hasta nada menos que Abril del año 93. ¡Más de un año después! Su último número incluía un mensaje del editor, encabezado por un gigantesco ¿Bye-Bye?, que a pesar de traer malas noticias (el final de la época comercial del Spectrum), ofrecía un claro mensaje de esperanza:

*"El Spectrum está abandonando su fase comercial para entrar en una fase especialmente interesante. Si te dedicas a él sobrevivirá"*.

Con un enfoque muy optimista, esta misma editorial anima a los usuarios a formar asociaciones locales y nacionales de usuarios por toda Gran Bretaña para preservar esta máquina en el futuro. En cierta forma es un espíritu parecido al que se desprendió en Septiembre de ese mismo año de la revista Your Sinclair, que incluyó un extenso reportaje sobre emuladores de Spectrum en los más variopintos sistemas, como el Amstrad CPC o el Sinclair QL, por dar un par de ejemplos. Pero describamos algunos de los juegos reseñados en el número final de Sinclair User, resultado de la unión de dicha revista con la difunta Crash Magazine.




*Nigel Mansell's World Championship*

La portada estaba dedicada a **Nigel Mansell's World Championship**, un simulador de conducción con una carga eterna. Los aburridos gráficos monocromos y los limitados efectos de sonido no lo hacían demasiado atractivo técnicamente. Por si fuera poco, la visibilidad es prácticamente nula; te enteras de que has entrado en una curva cuando tu coche se ha salido de la pista. A su favor tenemos unos controles simplificados que lo hacían mucho más asequible que otros programas similares, y un nivel de dificultad no tan elevado.

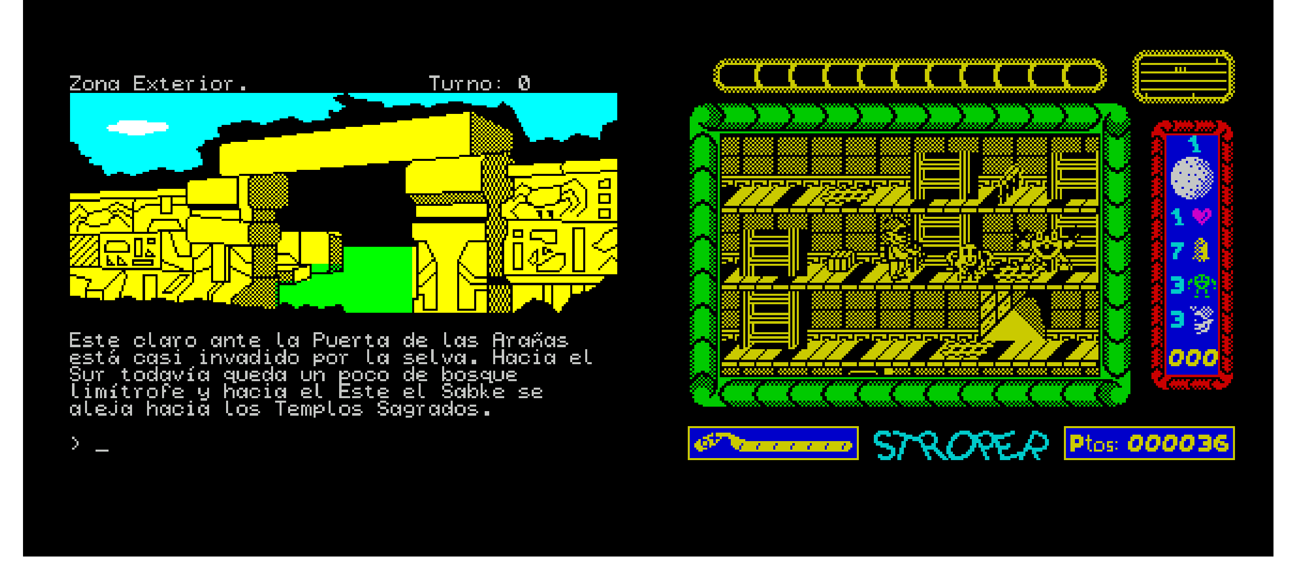
Otro título deportivo comentado en la revista es **World Rugby**, que no se debe confundir con aquel World Class Rugby del año 91 que sí pudimos ver en España. Se trata en esta ocasión de un juego de gestión de equipos en el que los partidos no son interactivos. Tan solo podemos contemplar las jugadas más relevantes sin hacer nada. No hay muchas opciones, el sonido es inexistente, y además las animaciones y los gráficos de los partidos son bastante pobres.




*World Rugby*

Siguiendo con los deportes, se revisan **Bully's Sporting Darts** y **R.B.I. Two Baseball**, dos productos sin mucha repercusión. También se habla del conocido por todos **Terminator 2**, que nos trae de nuevo el recuerdo de conversiones de otras películas como Robocop y Batman, que tan buen gusto nos dejaron. Quizá de lo mejor comentado en esa revista.

¿Qué fue del software español en esa época de oscuridad entre el último número de las revistas MicroHobby y Sincalir User? El año 92 no fue muy espectacular, que digamos. Si hacemos uso del buscador de World Of Spectrum podremos comprobar por nosotros mismos que tan solo aparecen dos títulos comerciales españoles durante dicho año. El primero de ellos es la última entrega de la trilogía Ci-U-Than, por aventuras AD: **Chichen Itza**. Una aventura conversacional que marcaría el final de la compañía, y que tampoco es de las más recordadas. El segundo es **Stroper**, de Zigurat, un plataformas bastante insulso. Un hecho curioso de ese año 92 fue la enorme proliferación de aventuras conversacionales españolas creadas por aficionados, tendencia que se mantuvo también en el año 93. El Professional Adventure Writing System, publicado unos años antes, fue sin duda un apoyo fundamental para el resurgimiento de este género en nuestro país, al menos a nivel amateur.




*Chichen Itza y Stropper*

La época comercial del Spectrum terminó, pero los aficionados mantuvieron viva a la máquina. Durante estos últimos años hemos asistido al lanzamiento de nuevos videojuegos programados por aficionados que, gracias a la inmensa cantidad de información disponible en la red, y a la que era imposible acceder durante la época comercial de nuestra añorada máquina, por fin han sido capaces de cumplir su sueño de infancia o juventud. La tendencia en estos juegos actuales podría ser descrita como ¿involución?, o lo que es lo mismo, la publicación de programas con unas características técnicas que bien podrían asemejarse a las de los juegos de los años anteriores al 90. Una vuelta atrás con gráficos y desarrollos simples. Alguien podría encontrar la causa de ese hecho en la inexperiencia de los programadores, que intentan a marchas forzadas aprender más y más con tal de comenzar de nuevo donde se quedaron las cosas. Pero personalmente prefiero pensar que la gente, en general, prefiere abandonar aquella locura de los años 91 y 92, y es más consciente de las limitaciones del Spectrum. Como consecuencia, los nuevos juegos son programados atendiendo a estas limitaciones.




*Infinity y Sokoban*

Es difícil anticipar qué es lo que va a suceder en el futuro. No sabemos si habrá personas decididas a seguir programando para los ordenadores Sinclair, ni tampoco cómo serán los nuevos programas. Tan solo esperemos que la nostalgia no nos impida ser críticos con las últimas producciones comerciales y que todos seamos capaces de apreciar aquellos nuevos juegos que en verdad sean divertidos sin necesidad de que nos intenten tan solo entrar por los ojos. A fin de cuentas, eso es lo que criticamos de los videojuegos modernos, ¿no?

Pablo Suau










# Programa aventuras en tu PC para Spectrum con InPAWS

*Recientemente ha sido editada una aplicación que puede hacer las delicias de aficionados al Spectrum que gusten de crear aventuras conversacionales. Esta pequeña maravilla ha sido bautizada por su creador con el nombre de **InPAWS** (muy probablemente en contraposición a la herramienta **UNPAWS**, que hace una tarea más o menos contraria a la que hace InPAWs).*

Para entender InPAWS primero tendremos que entender PAWS. Para aquellos que no lo conozcan, PAWS (Professional Adventure Writing System) es un sistema de creación de aventuras para Spectrum, Amstrad y PC, creado por Graeme Yeandle y Tim Gilberts, distribuido por la empresa Gilsoft. Sin duda la más famosa y usada de todas las versiones es la de Spectrum, con la que se han realizado joyas como "Jack the Ripper", la española "Abracadabra" o las más recientes aventuras, también españolas, de la saga del Dr. VanHalen.

PAWS permite programar una aventura sin tener conocimientos de programación, implementando un lenguaje sencillo basado en condactos (condiciones y actos).

PAWS

Crear aventuras hoy en día con PAWS se había vuelto más sencillo, al usar emuladores y permitir así grabar la aventuras según la avanzábamos con un simple snapshot. Sin embargo, con InPAWS, la facilidad es aún mayor, mucho mayor, al permitirnos escribir la aventura en nuestro PC, con un editor de textos cualquiera.

InPAWs toma un fichero de texto en el que se especifican todos los datos de una aventura, y genera con ello un fichero en formato .tap, que contiene la base de datos de la aventura tal y como la graba el PAW de Spectrum, permitiendo así que la aventura se cargue en el PAW de Spectrum como si lo hubiéramos hecho allí mismo.

Uno de los engorros de PAWS fue siempre el sistema numérico: en PAWS, cada localidad, cada objeto, cada mensaje, etc. estaban en una lista numerada, y cuando teníamos que hacer referencia al mismo, debíamos recordar el número que le correspondía. Esto hacía que todo usuario de PAWS tuviera a su lado su cuaderno, o

su hoja, con estas listas, para ser capaz de acordarse. Con la llegada de InPAWS sin embargo esto ya no es necesario, puesto que permite ponerle un nombre a una localidad al estilo “Cocina” o “Precipicio”, y el propio InPAWS se encargará de ponerle un numero cuando exporte para PAWS. Obviamente lo mismo se puede hacer con objetos o mensajes, de tal modo que es mas fácil recordar el objeto barra por el nombre “barra” que por el número “7”.

Adicionalmente, otro problema de PAWS ha sido siempre la total separación de cada tipo de datos en su editor: teníamos por un lado los mensajes, por otro las localidades, por otro las conexiones entre localidades, etc. Esto hacía que determinar todas las cosas relacionadas con un lugar, con un puzzle concreto, pudiera ser complicado. InPAWS sin embargo no marca prácticamente orden alguno, por lo que es perfectamente factible tener los mensajes relacionados con una localidad, o con un puzzle, incluso las condiciones o acciones a realizar en ese puzzle, juntos en el código.

Veamos algunas porciones de “código” tomadas de la Demo que acompaña a la primera release de INPAWS (no se muestra el fichero completo, tan sólo porciones de código que muestran cómo se programa con esta herramienta). Nótese que algunas líneas (largas) se han partido para facilitar la lectura del mismo:

```
//--- Declaración de Flags: --------------------------------------------------

// Flags usados en la aventura
FLAG portalonAbierto; // Portalon de la entrada
FLAG aldabonQuitado; // Si el jugador ha quitado el aldabón de la puerta
FLAG visitadoRecibidor; // SET la primera vez que se visita
                        // para que de un bonito mensaje


//--- Declaración de localidades: ---------------------------------------------

// -- Presentación y asignación de variables globales

LOCATION Inicio 0 // La localidad de inicio debe ser siempre la cero
{
   " De vuelta a casa, tras tu último trabajo en un prestigioso despacho de
abogados tomado por una horda de horribles vampiros chupasangres...es decir,
por \"otra\" horda de horribles vampiros chupasangres, recibes una llamada del
alcalde de XXX, un pueblo de los alrededores, para un nuevo trabajo.^^ Se
trata de investigar una serie de extraños sucesos y desapariciones en los
alredores de la vieja mansión junto al cementerio. Sin dudarlo aceptas el
trabajo, un nuevo reto para:^^{16}{2}        LIMPIEZAS FACUNDO^  EXORCISMO
Y DESVAMPIRIZACION^   SERVICIO EN TODO EL MUNDO";


//-------- Localidad de inicio del jugador
Location Verja

{
        " Te encuentras frente a la verja de entrada a la vieja mansión,
un imponente edificio de vetustos tejados ennegrecidos por el tiempo y
siniestros gabletes acechantes. La verja está abierta.";
        CONNECTIONS { N to Entrada ENTRA to Entrada };
}

   // Vocabulario y respuestas a comandos en esta localidad:
  VOCABULARY { Noun: "VERJA"; Noun: "MANSI", "EDIFI"; Noun: "GABLE"; }
  RESPONSE { EX VERJA: AT Verja MESSAGE "Oxidada." DONE;
             ABRE VERJA: AT Verja MES "Ya está abierta, además
                    e" MESSAGE "stá demasiado oxidada para moverla." DONE;
             CIERR VERJA: AT Verja MES "E" MESSAGE "stá demasiado oxidada
                        para moverla." DONE;
             EX MANSI: AT Verja MESSAGE "Tendrás que adentrarte para
                        conocer sus secretos..." DONE;
             EX GABLE: AT Verja MESSAGE "Puntiagudas formaciones que
                        te producen escalofríos." DONE;
             SALIR _: AT Verja MESSAGE "Un Robledillo jamás abandona un
                        trabajo sin acabar." DONE;
        }


//--- Localidades sencillas (descripciones y conexiones): ---------------------
```

```
Location PasilloSuperior
{
 " Un estrecho pasillo comunica las habitaciones y
la escalera principal que baja al salón.";
  CONNECTIONS { BAJAR to Salon SUR to PuertaPasillo Norte to FinPasillo};
}

Location FinPasillo
{
  " Te encuentras al final del pasillo, frente a un enorme ventanal que
te indica el final del camino.";
  CONNECTIONS { S to PasilloSuperior };
}

Location PuertaPasillo
{
  " Estás frente a la puerta de una habitación al final del pasillo.
  A diferencia de las otras puertas, esta parece que se conserva en
  un estado aceptable, sus goznes engrasados, su madera cuidada, el
  pomo intacto. La posibilidad de que alguien pueda habitar aún esta
  casa te produce cierto escalofrío.";
  CONNECTIONS { N to PasilloSuperior };
}


// La localidad 0 se presenta y tras esperar una tecla, saltamos a la
// localidad de nombre "Verja":
PROCESS 1
{
        * _: AT Inicio PROMPT 2 ANYKEY GOTO Verja DESC;
        _ _: NEWLINE PROCESS Salidas;
}


//--- Ejemplo de objetos: -----------------------------------------------------
Object abrigo
{
        "Un abrigo";
        INITIALLYAT WORN;
        WORDS ABRIG _;
        WEIGHT 1;
        PROPERTY CLOTHING;
}

VOCABULARY { NOUN: "abrig"; }
```

Otra ventaja de InPAWS es que mejora la compresión: para permitir la inclusión de más texto en los 48K del Spectrum, PAWS tenía un sistema de compresión basado en sustituir los códigos ASCII superiores a 127 y correspondientes a los tokens del Spectrum, con grupos de letras muy utilizados en la aventura. Para eso PAWS tenía su opción "Compress" que realizaba el análisis de los textos tratando de encontrar textos repetidos y substituía, por ejemplo "la" por un token, ocupando 1 byte en lugar de 2, y así con otras coincidencias, a veces de mas letras). InPAWS por su parte se aprovecha de la potencia de un PC para hacer lo mismo, pero consiguiendo mejoras en la compresión en la mayoría de los casos (probablemente dado que un PC puede hacer muchísimas mas "pasadas" en el mismo tiempo que un Spectrum, aunque esto, al no ser aún los fuentes públicos, es básicamente una conjetura del que escribe estas lineas).

Como todo no podía ser perfecto, InPAWS adolece de un problema: es incapaz de definir los gráficos de la aventura de una manera natural. Sin embargo, el propio InPAWS permite que tras generar dichos gráficos en el PAW, se puedan importar para ser usados desde InPAWS y así poder seguir usando InPAWS tranquilamente con esos gráficos.

La última versión de InPAWS conocida cuando escribo este artículo es la RC2, que además cuenta con una interesante capacidad: puede generar, además del fichero .tap para Spectrum, los ficheros fuente que usa el PAW de Amstrad (para CPM en realidad) y el PAW de PC, con lo cual es en teoría posible hacer con un solo fuente tres versiones del mismo juego.

El autor de InPAWS, Francisco Javier López, alias Mastodon, era hasta el momento desconocido en el mundillo tanto de las conversacionales como del Spectrum, pero eso va a acabar porque ha accedido a realizar una entrevista para MagazineZX, que podéis leer a continuación.

# Entrevista a Mastodon (Autor de INPAWS)

*Tenemos hoy con nosotros a Mastodon, autor de la herramienta InPAWS, algo que promete resucitar el viejo PAWS con aventuras para Spectrum (y Amstrad) hechas en nuestros PCs y sin saber programar.*

**MZX> Hola Mastodon**

Muy buenas.

**MZX> Bueno, antes de entrar en nada más, nos gustaría que te presentaras un poco para que te conozcan. ¿Quien es Mastodon? ¿De donde es? ¿Estudias o trabajas? ¿En que?**

Pues para empezar te diría que Mastodon es un personaje de una saga de aventuras creada por Steve Meretzky, del que me considero un admirador, y que simplemente me pareció gracioso como nick al darme de alta en algún foro. Si te refieres así en plan personal, pues para ser conciso y que luego no me recorten la entrevista te diré que tengo 37 años, que actualmente vivo en un pueblo de la provincia de Madrid y que trabajo en el sector de las telecomunicaciones (soy informático, PIM dicen algunos, je je), aunque mi labor profesional actual no guarda relación con la programación.

**MZX> Cuéntanos… ¿cuando conociste el PAWS? ¿Has realizado alguna aventura con él en otro tiempo?**

Mi primer contacto con PAW fue bastante precoz. Antes siquiera de que Samudio nos hiciera soñar con su peculiar estilo en Microhobby (que tiempos), llegó un día mi hermano por casa con una cinta pirata y unas fotocopias borrosas en inglés (siempre traía cosas raras, el muy cabrón). En la cinta había una versión del PAW en inglés. Yo por aquel entonces, aunque conocía las aventuras, no sabía que existían herramientas para crearlas ni que diablos era un "parser". Cuando lo vi funcionando recuerdo que me quedé bastante impactado, era sencillamente una maravilla. Estamos hablando posiblemente del año 1986 o el 87. Tiempo después hice una aventura con temática Western, que sinceramente espero que nunca vea la luz, además de algún proyecto de ciencia ficción inacabado.

**MZX> ¿Y el Spectrum?¿Como llegaste al mundo del Spectrum?**

Curiosamente el primer ordenador que pasó por mis manos fue un ZX81, de esos de un K y teclado pintado, que nos había prestado un amiguete. Luego se empezó a poner de moda el Spectrum así que terminamos pidiendo uno a los reyes. Finalmente nos trajeron un plus, de los que tenían teclado "profesional". En aquel momento casi todos nuestros amigos ya tenían Spectrum así que en ese sentido fuimos de los rezagados.

**MZX> ¿Conservas aquel Spectrum u otro? ¿Tienes algún entorno real funcionando o sólo emuladores?**

Lo conservo todo, incluso las cintas, aunque por razones prácticas sólo utilizo un emulador.

**MZX> ¿Ademas de programador has sido alguna vez jugador de aventuras?¿Tienes una favorita? ¿Como conociste las ACs?**

Sí, bueno, en el tema de las AC me considero más jugador que otra cosa. Mi primer contacto con las aventuras vino con juegos como Yenght, La princesa, o El hobbit, aunque por aquel entonces sólo era un granuliento "jugón" y para mi las aventuras eran simplemente un género más. Luego vino el PAW y la serie de artículos de Samudio, y entonces sí que empecé a interesarme más en serio por las aventuras, y menos por otro tipo de juegos. Con el advenimiento de Internet tuve la oportunidad de recuperar la afición, y desde ese momento vengo jugando con más o menos frecuencia a alguna que otra. Tengo muchas favoritas, aunque todas de la escena inglesa, ya que la española no la conozco mucho: Curses, Anchorhead, Sorcerer, Planetfall, Leather Goddesses of Phobos… en el Spectrum me gustaban las de Melbourne (Hobbit, El Señor de los Anillos) y sobre todo Level 9.

**MZX> ¿Que te parecen los interpretes de Melbourne o Level 9? ¿Hay algo en ellos que creas que el de PAW adolece?**

El más completo era el de Melbourne, aunque a mí siempre me han parecido más divertidos los juegos de Level 9. El principal problema de PAW era la falta de memoria, que obligaba en la mayoría de los casos a trocear los juegos en varias cargas.

**MZX> En cuanto a InPAWS, ¿como se te ocurrió hacer esta herramienta?**

Pues tenía (y tengo) un proyecto en mente, una especie de homenaje a Level 9, de hacer una aventura para Spectrum estilo Red Moon. Mi idea, no sé si es una locura, era que explotando las posibilidades de PAW haciendo uso intensivo de procesos, descripciones cortas y efectivas y puzzles sencillos de implementar, sería posible meter en 48k una aventura de unas 200 localidades con sus objetos, descripciones, personajes, e incluso gráficos. Lo de los gráficos no lo tenía tan claro, aunque quien haya visto los gráficos minimalistas de Level 9 comprenderá a que me refiero.

El caso es que incluso comencé el proyecto, tecleando la aventura en el PAW original, haciendo apuntes en un excel con los objetos, flags, etc. Cuando empecé a crear procesos, me vi apuntándolos también en el excel con cada uno de los condactos comentados. Total, no es difícil imaginar que me desanimé al cabo de poco tiempo.

Así que se me ocurrió que quizá podría hacer un programilla para convertir un fichero de texto con comentarios a una aventura de PAW. Se trataba, como planteamiento inicial, simplemente de poder comentar las líneas de

PAW, nada más. Así empezó todo.

**MZX> Cuéntanos un poco el desarrollo de InPAWS... ¿fue mucho tiempo?¿Empezaste alguna vez y lo dejaste o una vez que te pusiste fue todo de tirón? ¿has utilizado alguna herramienta aparte del compilador?**

Pues el desarrollo me ha llevado un par de meses aproximadamente. Varias veces estuve tentado de hacer una pausa, pero como me conozco demasiado bien sabía que esa pausa sería definitiva, así que me obligué a terminarlo. En algunos momentos me sentí un poco como Sísifo. Por ejemplo, cuando cargué la primera prueba en un emulador, no funcionaba. Todo estaba como en UNPAWS, estaba seguro, pero no funcionaba. Así que tuve que hacer mi propia labor arqueológica comparando byte por byte los tap de InPAWS con las originales para ver las diferencias. Algunos de los datos que se vuelcan a la memoria del tap no sé ni para qué valen, solo sé que están en todas las aventuras y que si no los pongo no funcionan. En otros casos tengo que volcar el número de localidades, o de objetos en cierta dirección de memoria además de en la que dice UNPAWS que están. Al final conseguí que las aventuras generadas por InPAWS fueran idénticas, byte a byte a las originales.

Otro tema gracioso fue el de la compresión. Con mi moderno PC sin limitaciones de memoria me pareció que me iba a comer literalmente al algoritmo compresor de PAW en 48k, y las aventuras de InPAWS serían infinitamente más cortas que las de PAW. Cuando compilé mi primera aventura con el nuevo algoritmo: ¡plof! Superada la memoria de PAW. Joder, que golpe a la soberbia. Así que tuve que afinar mucho, y digo mucho, para que las viejas aventuras siguieran entrando en la versión para la que fueron creadas, tras recompilarlas con InPAWS. Sabía que si InPAWS comprimía menos que PAW, aparte de decir muy poco de la utilidad de la herramienta, no animaría a la gente a utilizarla. Al final InPAWS comprime un poco más que PAW. También es de justicia dar un reconocimiento al buen hacer de Graeme Yeandle. Es un monstruo el tío.

Sobre las herramientas utilizadas, solamente he utilizado el compilador. Tampoco he tirado de ninguna librería aparte de la estándar de C++. Está todo hecho desde cero.

**MZX> ¿Qué te ha parecido cómo se ha recibido tu herramienta en los distintos sitios donde la has presentado?**

Pues creo que es algo pronto para opinar sobre eso. Me he anunciado en los foros donde pienso que puede haber interés, y la respuesta ha sido bastante positiva, aunque algo tímida. Soy consciente de que una herramienta de este tipo puede ser algo no muy demandado. Sin embargo, pienso que a los que todavía se interesan por sacar cosas retro les va a ayudar mucho. El caso es que existiendo cosas como Pasmo, BASIN, etc., está claro que InPAWS tiene su lugar dentro del mundillo retro-informático. Hay que darle tiempo.

**MZX> ¿Tienes algún plan para el futuro con InPAWS? ¿Crees que hay más cosas que hacer?**

Sí, hay bastantes cosas. Lo primero es que permita crear aventuras de 128k para completar su funcionalidad. También hay que mejorar el algoritmo de compresión y dar más flexibilidad al lenguaje. Otra línea de trabajo interesante es hacer un intérprete de aventuras PAW, al estilo del de Level 9 escrito por Glen Summers, pero con capacidades de depuración más allá de las que ofrece PAWS: me refiero a ejecución paso a paso, breakpoints, observación de flags (watches), etc. Lo que está claro es que para que merezca la pena meterse en estos proyectos tiene que haber interés, y cuando digo interés me refiero a algo más que considerar que la herramienta es útil. Me refiero a interés real en hacer cosas con ella.

**MZX> ¿Quieres decir hacer otro intérprete para Spectrum o para PC? Si es lo primero... ¿No crees que sería más sencillo, u otra opción, modificar la última versión de PAWS para darle más capacidades? ¿Quizá algunas capacidades que tienen los parsers herederos de PAWS como Superglus o SINTAC? ¿O a que te refieres exactamente con hacer un intérprete de aventuras al estilo del de Level9?**

No, no, tocar el PAW de Spectrum descartado, no sabría ni por donde empezar. Estaba pensando en "Level9", que es el nombre dado a un intérprete para PC escrito por Glen Summers y David Kinder a finales de los 90. Actualmente va por la versión 4, corre en varias plataformas y es capaz de ejecutar todas las aventuras de level9, incluyendo las de gráficos del Spectrum, aparte de otras posibilidades.

**MZX> ¿Piensas utilizar tu mismo InPAWS para crear una aventura?**

Sí, quiero retomar el proyecto de Red Moon 2 (o como se termine llamando), y también hacer alguna aventura con temática algo alejada de lo que habitualmente estamos acostumbrados a ver en PAWS. Algo más en la línea de la moderna ficción interactiva sin puzzles.

**MZX> ¿Tienes algún otro proyecto en mente? (relacionado o no relacionado con PAWS/Spectrum)?**

Montones de directorios huérfanos en mi disco duro, je je. Y varias aventuras empezadas con Inform.

**MZX> ¿Como ves el mundillo del Spectrum a día de hoy?**

Pues estoy bastante sorprendido. 25 años después de la aparición del Spectrum, y más de 15 años después de su "muerte", no sólo hay hay una labor de preservación monumental con páginas como WOS o SPA2, sino que la gente sigue sacando juegos para el Spectrum y hay aplicaciones de desarrollo realmente sorprendentes y complejas.

**MZX> ¿Y el de los conversacionales?**

Creo que goza de muy buena salud, aunque a veces sufra pequeños trastornos mentales como Inform 7, que

con suerte se curarán pronto. Únicamente me da la impresión (y digo la impresión por que no conozco demasiado el tema) que en España estamos algo rezagados en cuanto a cantidad, de calidad seguro que no hay nada que envidiar. Es una pena, aunque también es verdad que si tipos como yo sacaran alguna cosilla en vez de quejarse, contribuiríamos a equilibrar la balanza.

**MZX> Para terminar, si quieres añadir algo, estas líneas son tuyas:**

Simplemente animar a la gente a que siga haciendo cosas para el Spectrum (sea con InPAWS o no),para preservar la memoria de ese cacharro que tantos buenos recuerdos nos trae a algunos.

**LINKS**

- Página de InPAWS: http://inpaws.speccy.org/
- Página de Graeme Yeandle: http://www.yeandle.plus.com/advent/
- PAWS en WikiCAAD: http://www.wikicaad.net/PAWS
- WikiCAAD: http://www.wikicaad.net/ fuente de la portada del PAWS mostrado en este artículo.

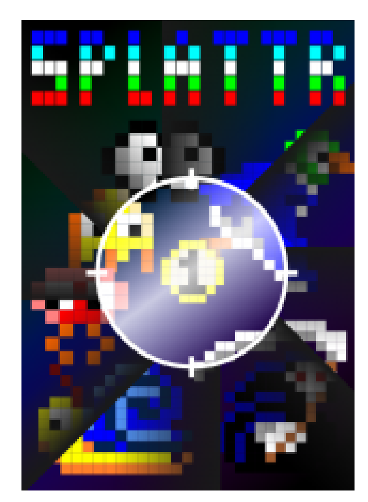
Carlos Sánchez



En este número, disponéis de los siguientes análisis:

- Albert Valls analiza tres juegos aparecidos durante el pasado año: splATTR, Rallybug y Nipik 2.
- Carlos Sánchez analiza, desde el punto de vista histórico, el mítico The Hobbit.
- Santiago Romero comenta uno de los clásicos del Spectrum: Flying Shark.
- Javier Vispe analiza el juego de CEZGS I need speed, el lanzamiento de World XXI Soft, Escuela de Ladrones y la Trilogía Farmer Jack.

**splATTR**




| | |
|---|---|
| Título | splATTR |
| Género | Acción |
| Año | 2008 |
| Máquina | 128K |
| Jugadores | 1 jugador |
| Compañía | Cronosoft |
| Autor | Bob Smith, Lee du-Caine |
| Otros comentarios | |

- Web Oficial

El factor riesgo en cuanto al uso de gráficos en baja resolución bien resueltos encuentra, desde los atávicos inicios del Z80, desenlaces de diversa índole.

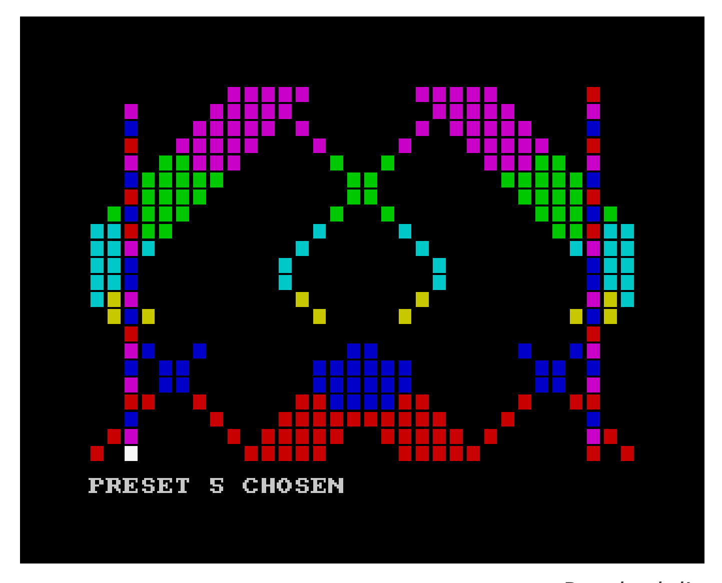



*3D Monster Maze*

De intentos los ha habido que, por su naturaleza y soporte en el cual se desarrollaron, se han ido elevando a la categoría de rompedores aun no habiendo dilema posible a la hora de escoger (se corrobora lo dicho tanteando el actual guarismo de devotos de algunos de los trabajos de J.K. Greye para ZX81). Llegado ya el ZX Spectrum la propia entidad de algunos programas sirve de perfecta coartada para el uso de predefinidos, especialmente en adaptaciones de juegos de mesa tipo Yahtzee u Othello, y sobre todo en rompecabezas (Nowotnik Puzzle, The Thinker o Flippit entre otros). Hablamos, pues, de la utilización de los atributos no como recurso... sino más bien como último recurso.

Para el campo arcade se hace más dificultoso encontrar precedentes, digámoslo así, puros. Prácticamente desde el primer momento se hizo uso de menos o más embellecedores para trabajos cuya esencia residía en la simplicidad gráfica de su desarrollo: motivos acogidos a standards que van desde Breakout hasta el mismo Tetris, pasando por Amidar (aunque en este último caso y también a modo de muestra, ni eso intentaron los señores de New Generation Software con su lamentable Road Runner). Otra cosa resultaría (y para nada procede ahondar en ello) si empezamos a repasar algunos títulos third-party de la fastuosa Cassette 50 de Cascade, en los que nada hay de malo en su elaboración (mayoritariamente amateur), como sí en su comercialización...

Obviamente no acontece, en ninguno de los anteriores ejemplos mentados, la premisa que Bob Smith sí decidió seguir para splATTR: imponerse esta limitación gráfica y proceder a su máxima explotación posible. Quizá Jeff Minter tuviera en su día un propósito vagamente parecido con su Polybius particular (hablo de Psychedelia, y permítaseme lo inexacto de la aproximación), si bien en su caso las dosis de lisergia doméstica no resultaron especialmente bien acogidas (y a fe que lo notaron en Llamasoft).



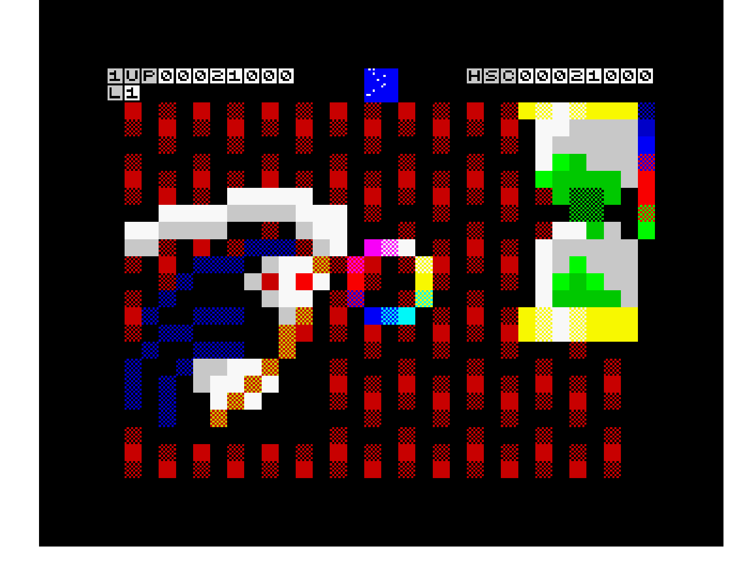

*Psychedelia*

Se empezó a saber de splATTR a mediados de 2007 en tanto su autor fue mostrando su evolución y primeras beta entre los asiduos de los WoS forums. Ello permite, ya en esas ocasiones, dar cuenta por parte de éstos del factor más novedoso del juego; a la par que al propio Bob el particular trabajo de campo le sirve para escuchar y poner en práctica algunas de las sugerencias surgidas en dichos intercambios de opinión (de las cuales más de una llegó directamente desde tierras aragonesas...). Foto (¿anecdótica?) aparte para la espontanea cuota de proposiciones de bautismo para el juego, que queda cerrada en fecha concreta (16 de julio del mismo año) gracias al ingenio de Andrew Owen. En adelante, ese 'project ATTR' (utilizado hasta entonces como working title) pasa a tener apelativo definitivo.

splATTR puede resultar, con el tiempo, un más que acertado paradigma del attribute-based game actualizado y con intención, y que intenta aprovechar conocimientos de programación que el tiempo y el presente acceso a la información permiten ya exigir. Sirva aquello de que más vale poco y bien, antes que el deseo de abarcar mucho para no llegar a nada: se parte de una idea, se trabaja en ella y se resuelve de manera correcta. Siendo éste el principal motivante (el desarrollo de un engine específico para uso avanzado de atributos), Bob sirve el plato en forma bastante más común: un shooter plano y con visor zenital, según el cual deberemos ir eliminando a los variopintos meanies que vayan apareciendo en pantalla de cara a procurarnos el avance a fases posteriores.

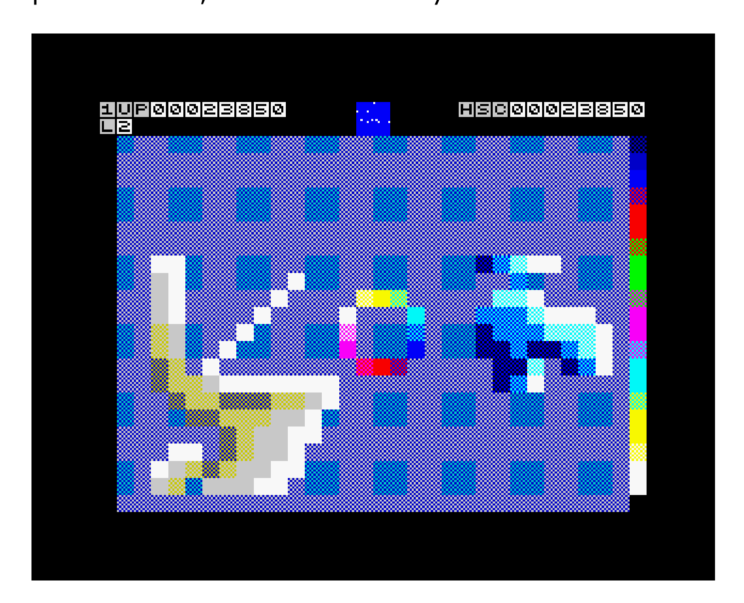
Una vez más el autor opta por su estilo de menú de inicio 'a la Panayi'. Gratamente a dicho menú lo acompaña esta vez un explícito tributo a John Conway: en segundo plano van sucesivamente circulando diversas series de planeadores, osciladores, bloques y demás fauna de común encuentro en el conocido Juego de la vida (cuatro LWSS dan la bienvenida nada más terminar la carga del programa). Opten por hacer revisión de la décima edición de la presente publicación y del artículo que firma Santiago Romero, si sintieran que en este último párrafo les he empezado a hablar en gulevache...

Tanto aquí como en el screen$ previo tenemos una primera muestra de lo que representa uno de los más importantes recursos usados por Bob. El optical dithering no es algo precisamente nuevo en Spectrum: vistosas muestras de su uso las encontramos en la demoscene (vayan recomendados grupos como Noumenon o Brainwave), ya desde los años 90 y hasta nuestras fechas; y trascendiendo el abanico de software comercializado, basta con referirse al manual de Basic del propio ordenador, en el que ya se intentaba mostrar el potencial del UDG tipo rejilla a modo de tablero de ajedrez para "obtención" de colores alternativos a los existentes por defecto. En splATTR, no obstante, su presencia pasa a ser una herramienta constante y permite retornar la vistosidad cromática de la que gozan la gran mayoría de personajes.




Dejando de lado las habituales opciones en cuanto a control, el juego permite escoger entre un fondo animado o estático, así como el tipo de seguimiento que podemos efectuar sobre los enemigos presentes en los escenarios. Disyuntiva al canto entre un radar basado en los ejes vertical y horizontal (mostrados en los laterales de la pantalla), o bien uno de visión reducida que abarca sólo parcialmente el entorno de nuestra posición. A ambas opciones de juego volveré más tarde.

El programa dispone de un número de escenarios a recorrer variable, en función del nivel de dificultad escogido. Dicho recorrido, en cualquier caso, está diseñado mediante celdas que se muestran dispuestas en forma de diamante, cada una de ellas siendo un escenario distinto (metodología similar, solo que en forma piramidal, encontrábamos en consabidos clásicos como Out Run y, oh sorpresa, Pyramid). De esta forma, y en trayectoria descendente, el jugador escoge su ruta siempre que los límites de ese diseño así se lo permitan. Encontramos, pues, un número de fases y posibles escenarios distintos para cada skill (5 y 9 para el fácil, 9 y 25 para el mediano, y 15 y 64 para el difícil; ¿han adivinado ya sendas sucesiones?).




Con cada celda se muestra una leyenda que nos indica el objetivo a cumplir. A partir de aquí, el propósito es siempre el mismo: "limpiar" cada una de las areas para avanzar hacia la siguiente. Las circunstancias de juego son diversas: debemos eliminar enemigos móviles, o bien hacernos con los goodies (objetos a modo de bonus) que aparezcan, sean móviles o estáticos. A su vez nos encontramos con obstáculos de diversa ralea implementados en según qué tipo de escenario: enemigos que nos persiguen, muros, laberintos a resolver, scroll forzado (en idéntica forma que en Splat!), etc. A medida que uno se familiariza con cada escenario (hay 32 distintos en total) se hace interesante ver la variedad de patrones de movimiento que en splATTR vamos a encontrar; y, por tanto, la correspondiente cantidad de rutinas precisas a ejercer por nuestra parte en cada uno de los casos.

Contamos con un total de tres intentos, cada uno de los cuales se desvanece al agotar nuestra barra de energía. Ésta irá disminuyendo siempre que entremos en contacto con los enemigos o con los obstáculos presentes; no así con los bonus, que actúan en sentido contrario y, aunque en menor cuantía, la palían. La obtención de una mejor puntuación dependerá, en parte, de los riesgos tomados para con cada enemigo (su tipología dará a entender a cuáles de sus zonas hay que acertar para obtener más alta recompensa).

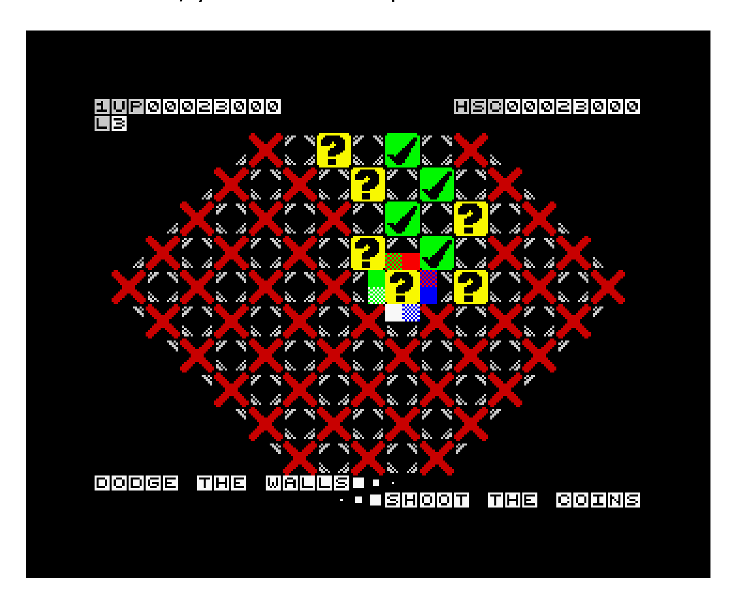
Pocas vueltas más hay que dar para mostrar los puntos fuertes de splATTR. Recurro de nuevo a hablar de una gran corrección en referencia a la puesta en acto del objetivo fijado. No es, desde luego, el primer programa para Spectrum que se apoya en el encanto de lo elemental mediante brillante ejecución (incluso en nuestros días; y a Alien 99 me es fácil hacer mención); pero sí significa un precedente por la forma en que lo consigue.

Las sucesivas partidas dejan entrever, casi con la misma sencillez, algunos 'debe' que hacen cohibir el elogio

desenfrenado y dan testimonio de algunas de sus carencias o posibles mejoras. Y es la primera de ellas la limitada longevidad que el juego ofrece. Sin duda hay heterogeneidad suficiente en cuestión de enemigos y entornos como para encontrar en splATTR numerosos ratos de diversión... pero la acción propiamente dicha no es tan variada. Algo de lo que seguramente adolece todo juego de desarrollo lineal, todo sea dicho. Búsquense las semejanzas, pongamos por caso, con 3D Tunnel o con Operation Wolf para poder hablar de splATTR como un juego mucho más refinado (y valga el calificativo en más de un sentido), aunque no por ello mejorado en cuanto a llano entretenimiento.

El indicador anterior hubiera podido salir mejor parado si en los escenarios encontráramos motivantes fijos que alejaran al jugador de la pasividad y el menor uso de reflejos. Nos podemos encontrar vagando indefinidamente o disparando, como comúnmente se dice, 'a bulto' sin que por ello se vea afectado el devenir de la partida en juego; y de haber existido (especialmente para algunos tipos de enemigos, o para escenarios como los laberintos) elementos de presión como una cuenta atrás o un número limitado de disparos, la carga de entrega del jugador se hubiera beneficiado muchísimo de ello.

En cuanto a los fondos y ayudas de posicionamiento, el balance no es precisamente positivo. Quizás ya de entrada la elección del tipo y ubicación de los radares tenía mala resolución, y por ello cabe no ser crítico con tanta severidad; pero ambos exigen al jugador mucha habituación a su uso (y especialmente el basado en los ejes). Peor resultan algunos de los fondos escogidos (apenas unos pocos, afortunadamente) que, se encuentren en estático o en dinámico, nos dificultan terriblemente saber dónde nos encontramos (nos movemos, aunque carecemos de tal sensación por falta de puntos de referencia), llevando a peligrosas desubicaciones. Y dado el funcionamiento particular del juego éste es un detalle que no debe sino tenerse muy en cuenta: recordemos que, siendo cada escenario notablemente mayor que nuestra pantalla, es dicho escenario quien "se mueve con nosotros", y no nosotros por él...




Cerrando aspectos de jugabilidad, dejo a valorar muy subjetivamente el grado de confortabilidad que el gran tamaño de los items y personajes incluídos ofrezca al jugador. Resultándome a mí suficientemente cómodo, los hay que manifiestan su cierto desagrado; en especial cuando el ratio de objetos en pantalla pinta un pelín exagerado.

Una vez más, Lee du-Caine repite en labores de composición y efectos sonoros. Melodías distintas para el menú de inicio, final del juego y distintas transiciones de escenario, aunque contrariamente a lo que se tuvo a bien con Stranded 2.5 el in-game ambiental es fijo. Uso muy variado y elegante de los distintos tipos de caídas de onda prioritariamente en las voces armónicas, y balsámicas licencias en cuanto a la partitura en sí (hoy día se agradecen trabajos que huyan del grado conjunto y de ciertos patrones de progresión de acordes más que quemados).

splATTR es distribuído vía Cronosoft en sendos formatos (.tzx y soporte físico a sus respectivos precios), obteniendo ambos por la compra del segundo. Vaya un toque de atención final para Simon (Ullyatt), cabeza visible del ente en cuestión, y al cual se le debería pedir un mejor acabado en algunos de los últimos títulos que ofrece. Y, por supuesto, una congratulación (y no precisamente protocolaria) para Bob Smith, teniendo en mente que lo mejorable no quita en absoluto lo más que notable de su última creación.

Valoraciones

| | |
|---|---|
| Originalidad: | [8] |
| Gráficos: | [9] |
| Sonido: | [8] |
| Jugabilidad: | [6] |
| Adicción: | [6] |
| Dificultad: | [6] |

**Flying Shark!**




| | |
|---|---|
| Título | Flying Shark |
| Género | Shoot'em-up |
| Año | 1987 |
| Máquina | 48K/128K |
| Jugadores | 1 ó 2 jugadores (alternados) |
| Compañía | Firebird Software Ltd y Dro Soft. |
| Autor | Dominic Robinson -Programación-, John Cumming -Gráficos- y Steve Turner -Sonido- |
| Otros comentarios | |

- Ficha en WOS

Flying Shark es una de las mayores joyas del ZX Spectrum. Esta magnífica conversión en 48K de la recreativa arcade de Taito es, sorprendentemente, desconocida por muchos usuarios del microordenador de Sinclair.

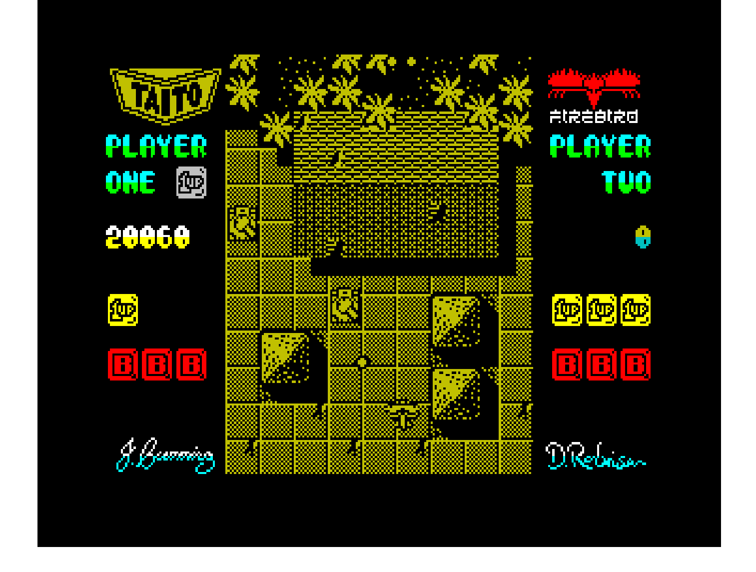
Acostumbrados a asociar las típicas glorias Spectrumeras a títulos como Head Over Heels, La Abadía del Crimen, Manic Miner, MatchDay y compañía, muchos jugones olvidan el género de los shoot'em-ups, que alcanza con juegos como Flying Shark unos niveles de calidad difícilmente mejorables con la limitada potencia del Spectrum. Flying Shark viene de la mano de Dominic Robinson, autor de otros juegos del género como Uridium o Zynaps, y de otros juegos de una gran factura técnica como Exolon.




*Pantalla de carga de Flying Shark*

El argumento del juego es el siguiente: has sido seleccionado por tus superiores, como as del vuelo y del combate que eres, para pilotar una misión casi suicida, tratando de llegar hasta el corazón de las tropas enemigas.

Bajo este sencillo argumento se esconde un shoot'em-up vertical donde controlaremos una pequeña nave que debe avanzar eliminando todos los enemigos que se crucen en su camino. Para ello contaremos con 3 vidas (representadas en el marcador con el icono de 1up), y en cada una de ellas podremos disfrutar de 3 bombas inteligentes, las cuales eliminan a todos los enemigos presentes en la pantalla, o infringen grandes daños a los enemigos finales contra los que estemos luchando.




*Flying Shark*

Nuestra nave puede disparar ilimitadas ráfagas de disparos cuya potencia podremos aumentar recogiendo los items con forma de "S", que doblan o triplican la anchura de nuestro fuego. Estos items aparecerán en pantalla cuando destruyamos todas las naves de algunas de las formaciones que nos atacarán, y son vitales contra determinados enemigos, como las torretas fijas.

Asímismo, algunos de los enemigos dejarán bonus en forma de bombas inteligentes adicionales (icono "B"), aunque sólo podremos llevar 3 de ellas en nuestro avión, por lo que si ya cargamos 3 de ellas y aparece una para recoger en pantalla, podemos aprovechar para lanzar una de ellas y volver a recargarla.

Cada enemigo abatido proporciona puntos, y en algunos momentos veremos que el icono de 1UP parpadeará, lo que nos proporcionará puntos adicionales si abatimos todos los enemigos de la pantalla mientras este símbolo parpadea. Obtendremos la primera vida extra a partir de los 50.000 puntos, la siguiente a los 150.000 puntos, y a partir de ahí cada 150.000 puntos obtenidos. Otra ayuda para conseguir alcanzar estas cifras son los bonus de 1000 puntos que aparecerán al destruir determinados enemigos o formaciones, aunque también podremos conseguir vidas recogiendo los escasos símbolos de 1up que aparecerán.




*Items S: más vale que los cojas*

Los controles son sencillos: 4 direcciones de movimiento para nuestro avión y un botón de disparo, el de las ráfagas de ametralladora. El mismo botón de disparo sirve para lanzar las bombas inteligentes si lo mantenemos pulsado más tiempo del necesario para disparar. Cuidado con esto porque, más de una vez en medio del fragor de la batalla podemos lanzar una bomba por error si no mantenemos el "tempo" necesario para lanzar ráfagas largas.

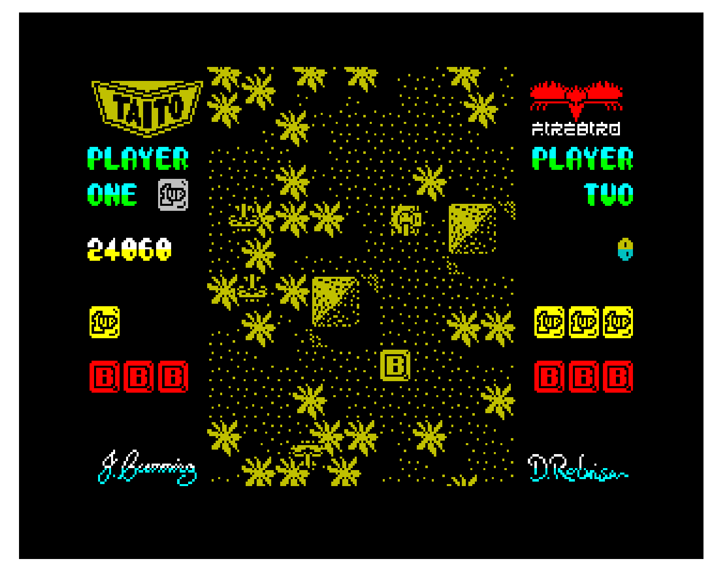
En la pantalla de inicio podemos utilizar las siguientes teclas:

- Mantener pulsado 1 ó 2: seleccionar modo de 1 ó 2 jugadores (alternándose sobre el teclado).

- Mantener pulsado 3: Redefinir el teclado.
- Disparo (teclado o joystick): Iniciar partida.

En cuanto a mapeado, el juego se desarrolla en 4 niveles:

En el primero, emplazado en la jungla, aparecerán aviones que se derriban fácilmente con un sólo disparo, así como tanques que requieren 2 disparos para ser eliminados (el primero de ellos eliminará su torreta, mientras que el segundo destruirá el vehículo en sí). También encontraremos emplazamientos fijos sobre el terreno de los cuales pueden salir vehículos y que nos dispararán continuadamente hasta que los destruyamos (cosa que, por otra parte, requerirá gran cantidad de disparos). Ojo con los aviones enemigos porque, obviamente, no sólo nos derribarán sus balas sino también perderemos una vida si colisionamos con ellos. Los tanques, sin embargo, pueden ser sobrevolados sin causarnos daño (siempre que no disparen), y en ocasiones bastará con inhabilitar su torreta de disparo para poder ocuparnos de otros enemigos que nos acechen.




*Pantalla 1: la jungla*

En el segundo y el tercero, que transcurren sobre el océano, nos enfrentará tanto a aviones como a barcos con torretas. Como descubriréis, es importante destruir todas las torretas conforme aparecen, ya que la forma en que están colocadas (verticalmente) requiere destruir cada una de ellas para poder atacar a la siguiente. El segundo nivel es el más largo del juego, claramente diseñado para esquilmar el número de vidas del jugador confiado después de finalizar el primer nivel. Os aseguramos que llegaréis a odiar las torretas de estos 2 niveles.




*Las odiosas torretas del nivel 2*

En el cuarto nivel, sobrevolaremos vías de tren y edificios, enfrentándonos a tanques y aeroplanos. Este es el último nivel del juego y necesitaremos aprovechar en él todas las vidas extra que hayamos conseguido, y las bombas inteligentes mantenidas. Notaréis (en realidad, ya desde el nivel 2) cómo los enemigos disparan mayor cantidad de balas y éstas se dirigen con más precisión y velocidad hacia vosotros.

En las partes finales de cada mapeado nos encontraremos a enemigos de "final de fase", los típicos enemigos "no estándar" que requerirán una mayor cantidad de disparos para ser abatidos. Por ejemplo, en el nivel 1 nos encontraremos un enorme tanque. Estos requerirán más de una vez que lancemos alguna bomba inteligente para reducir la resistencia del enemigo y requerir menos tiempo de disparo y maniobras de evasión en su destrucción.




*Tanque del final del nivel 1*

Cabe destacar que las trayectorias y puntos de aparición de los enemigos, como en todo shoot'em-up que se precie, son fijos, premiando al jugador que se tome la molestia de memorizar por dónde aparecerán las ráfagas de enemigos. Veréis como, después de un par de partidas, de forma instintiva vais colocando el avión por donde aparecerán los siguientes enemigos para abatirlos nada más aparezcan en pantalla.

A nivel gráfico, el juego muestra unos marcadores coloridos que contrastan con el desarrollo "monocolor" del área de juego. Y decimos "área de juego" porque emulando el formato "vertical" de la coin-op original (y, por qué no, ahorrando gran cantidad de tiempo necesario de proceso), la zona donde se desarrolla la acción apenas representa el 50% de la pantalla. No obstante, esto no supone un impedimento para el desarrollo y permite un scroll y movimientos de los enemigos y balas bastante fluído, mejorando la jugabilidad.




*Los sprites se distinguen mejor en las fases 2 y 3*

El apartado sonoro es típico de muchos juegos de 48K: música en los menúes y efectos sonoros durante el juego. La música es bastante buena y pegadiza (recordemos que es un juego para 48K) y los efectos simplemente cumplen.

Pero lo más importante del juego es, sin duda, la excelente combinación de jugabilidad, adicción y dificultad. Un juego difícil unido a un buen control y movimiento fluído dan lugar a que Flying Shark sea realmente adictivo. Repetirás una y otra vez, mejorando en cada partida puesto que, poco a poco, irás aprendiendo el lugar de aparición de las hordas enemigas y la mejor forma de encarar cada enemigo de mitad o final de nivel.

Valoraciones

Originalidad: [4]
Gráficos: [7]
Sonido: [7]
Jugabilidad: [9]

| | |
|---|---|
| Adicción: | [9] |
| Dificultad: | [9] |

**Trucos**

POKE 54462, 201 = vidas infinitas
POKE 53920, n = número de vidas
POKE 53962, n = fase inicial
POKE 60429, 0
POKE 60430, 0
POKE 60431, 0 = bombas infinitas

Santiago Romero

**Rallybug**




| | |
|---|---|
| Título | Rallybug |
| Género | Acción |
| Año | 2008 |
| Máquina | 48K/128K |
| Jugadores | 1 jugador |
| Compañía | N/A |
| Autor | Jonathan Cauldwell, Yerzmyey |
| Otros comentarios | |

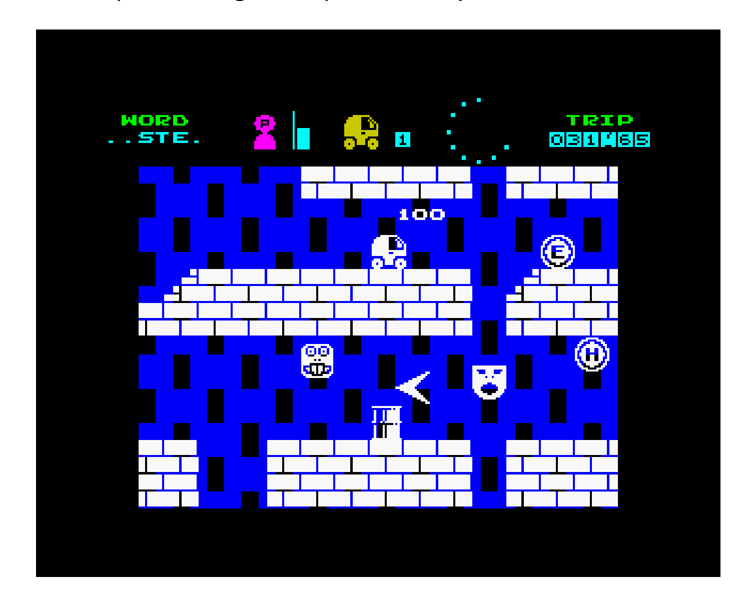
- Sitio de Jonathan Cauldwell
- Archivo .rzx de Rallybug
- Retrofusion
- Everyman

Segundo evento del año organizado por los britanicotes de Retro Fusion, ente de aficionados/usuarios que desde el año 2005 ha ido consiguiendo propósitos de intereses varios y simultaneos:

- la aclimatación (es un decir) de espacios para entregada reunión de afines al entretenimiento por video en sus múltiples variedades;
- el aunamiento de las más atávicas versiones de esa categoría (¿les hablo un poco de los pachinkos, o puede que no haga falta?) con las plataformas de más reciente factura mediante su uso, disfrute y escucha recíproca de saberes ajenos;
- y de propina, una causa paralela por la que sentir justificado el gasto de carburante, pedal o suela: el afortunado asistente es sabedor, al comprar su entrada o deglutir su cerveza, de su colaboración indirecta con Everyman, organismo implicado en la investigación y divulgación de la lucha contra el cáncer prostático y testicular. Créanme, qué mejor ocasión (ésta última de la que les hablo, y toda en su conjunto) para afianzar al unísono la vivencia de tan amistoso encuentro y la convicción de haber luchado en estimable actitud por tenerlos bien puestos...

Para antes de esos 19 y 20 de julio del año en curso, el señor Jonathan Cauldwell ya tenía su camiseta negra, motivo Egghead en pecho, convenientemente planchada. De tal (ojo, más que noble) guisa acudió a esta

Fusion '08 con el añadido de ofrecer, a quien gustase degustarlos, unos buenos vistazos y alguna que otra partida a su último trabajo para Spectrum (y del cual ha derivado este Rallybug que aquí se comenta). Parafraseando sus propias palabras de presentación, un acercamiento de comunión entre el Moon Alert al uso y un Great Giana Sisters (que nunca llegamos a probar bajo carcasa Sinclair alguna). Qué bien; por ahorrarme, hasta me he ahorrado referirme al tipo de la gorra que vdes. ya saben…




Y la trama, sencilla como se pueden imaginar: recolección (cuanto más completa mejor) de las banderas desperdigadas por cada uno de los 12 recorridos del juego (tres distintos niveles, cada uno a su vez con cuatro areas); procurando el sorteo de todo tipo de enemigos, así como evitando las caídas en falso llano y el agotamiento total del tiempo ofrecido para completar cada ruta.

Cauldwell, como ya decía, tuvo en consideración alguna de las sugerencias de mejora/expansión que le llegaron, y ya a finales de agosto presentaba la versión a priori definitiva del juego. ¿Novedades explícitas? Contamos con un nuevo nivel (y por tanto, las áreas contempladas pasan a ser 16); pero sobre todo, contamos con una nueva modalidad de juego.

En efecto. Desde el menú inicial podemos escoger entre la opción 'Fusion' (la original y ya descrita) y la 'Hangman', que cambia banderitas por letras. La recolección de éstas nos ha de permitir, finalizando cada area, ir solventando palabras de seis letras cual juego del ahorcado, y de esta forma repostar hasta arriba nuestro combustible (el cual va menguando, como es obvio, a base de kilometraje rodado). La no resolución a tiempo de estas palabras implicará que llenar nuestros tanques resulte, digamos, un poco más caro (pérdida de una de las vidas). Dejar resuelto cada nivel nos aporta, por el contrario, otro miss adicional con el cual contar.

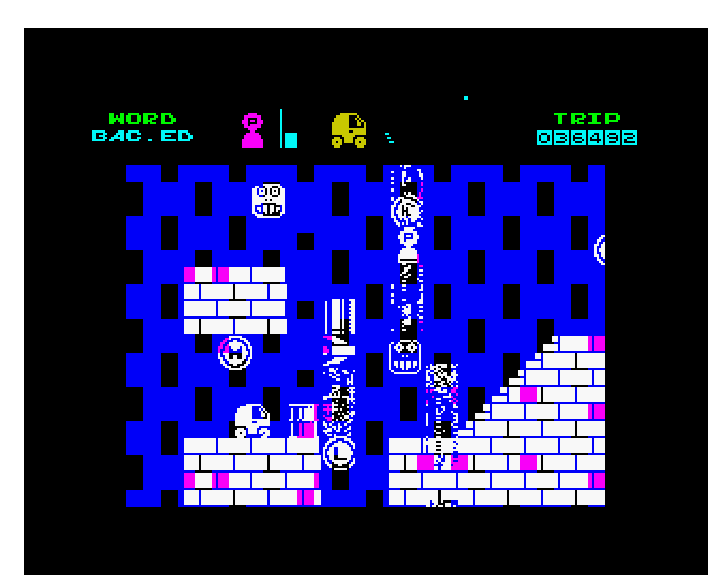
Sea por medio de una u otra opción, la dificultad del juego en escasos momentos resulta intimidatoria. A lo sumo se puede afirmar que las dos áreas posteriores de cada nivel ofrecen más variedad de enemigos molestos (y por ahí vendría la gradación, aunque sin temor la pueden perfectamente dejar en un sui generis). La asimilación adecuada de Rallybug pasa por un paulatino aprendizaje de escenarios, y un oportuno balance entre elección de mejores rutas y rapidez de resolución. De hacer recuento, no habrá en realidad tantos movimientos que deban atenerse al pixel-perfect como en un principio pueda parecer.

Llegados a este punto, no hay más remedio que soltar ya la de arena. Y es que se me hace más que evidente que urgen homologaciones en casa Cauldwell… porque me es harto difícil recordar algún otro “release” que llegue a devolver más errores que éste: a poco que uno pretenda forzar algún movimiento insólito, una ruta menos accesible que otras o bien un salto demasiado elevado, se expone al riesgo de corromper el juego en más o menos mesura. Los gazapos experimentables son múltiples: bailoteo de atributos imprevistos en pantalla, coche enterrado en el suelo o bien transitando por entre los ladrillos, gráficos y caminos completamente desplazados de lugar, elevadores que cobran velocidades de puro vértigo, funcionamiento erróneo del menú en modelos +2 y 128K… Especialmente recordables son el par que a continuación les transmito:




- la linea de llegada en el décimo nivel (3B) puede ser cruzada por encima sin que el programa dé por finalizado el recorrido. Con que el scroll del juego siga su curso no podremos terminar como es debido.
- en modalidad 'Fusion', nuestro auto sufre periódicos parones por quedarnos con el depósito vacío. Notable sinsentido, teniendo presente que en esta modalidad no se tiene en cuenta el factor combustible. En el RZX Archive pueden encontrar oportuna muestra de estas absurdas (y molestas) pérdidas de vida.
- Como excepción, eso sí, debe observarse una última circunstancia que podría pasar por ser una pifia más, sin serlo: cruzar la linea de llegada no siempre implica poder apartar las manos del teclado; en esos pocos segundos también algún enemigo puede dejarnos fritos. Si soy sincero, este tipo de “twists” me parece más que interesante…

Informado Jonathan hace unos días del cúmulo de funcionamientos anómalos, dejó en su 'to do' particular la futura resolución de los mismos, aunque sin plazo mínimamente previsible. La compilación de posibles mejoras que en su momento le fueron sugeridas poco menos que pasan indefectiblemente a un segundo plano: la mera resolución de tan considerables bugs (que en ocasiones llevan al bloqueo total, o al reseteo sin contemplaciones) sería el único, el mejor y el más deseable de los updates que puedan venir (si vienen).




Por lo demás, palidez general en cuanto a entorno gráfico y sonoro. Monocromía asociada a distintos fondos,

vía combinación/chute de atributo mondo y lirondo (recuerden Higgledy Piggledy y vayan avanzando, para mayor referencia). Y pocas alegrías por lo que respecta a la cuestión musical (ninguna, de hecho, en modo 48K); una única melodía presente (menú), en la que Yerzmyey juega relativamente bien, eso sí, con el uso de armónicos para presentar algunos de los fraseos melódicos, tal y como ha acostumbrado en sus asomos para la demoscene.

Rallybug, tal cual. Con su pertinente grado de adicción y jugabilidad que no podría dejar de constatar, Nuestro Señor me libre. A mis ojos, poco más o menos tan meritorio como el recóndito tributo a Kenny Everett que Jonathan se ha permitido por entre las entrañas del código. Todo sea por dejar un mínimo buen regusto a la revisión.

| Valoraciones | |
|---|---|
| Originalidad: | [2] |
| Gráficos: | [4] |
| Sonido: | [5] |
| Jugabilidad: | [6] |
| Adicción: | [7] |
| Dificultad: | [5] |

Albert Valls

**I need speed**




| | |
|---|---|
| Título | I need speed |
| Género | Conducción |
| Año | 2009 |
| Máquina | 48K/128K |
| Jugadores | 1 ó 2 jugadores |
| Compañía | Computer EmuZone Games Studio |
| Autor | Jaime Tejedor, José ignacio Ramos, Angel Lo, “Riskej”, Javier Peña. |
| Otros comentarios | |

- Descarga

I need speed supone la vuelta al género de las carreras en el Spectrum después de muchos años. Con un nombre que nos retrotrae a cierta saga comercial que todos conocemos, nos encontramos ante la ópera prima en ensamblador de Metalbrain. Enfundémonos el casco, encendamos motores y aspiremos el olor a gasolina.

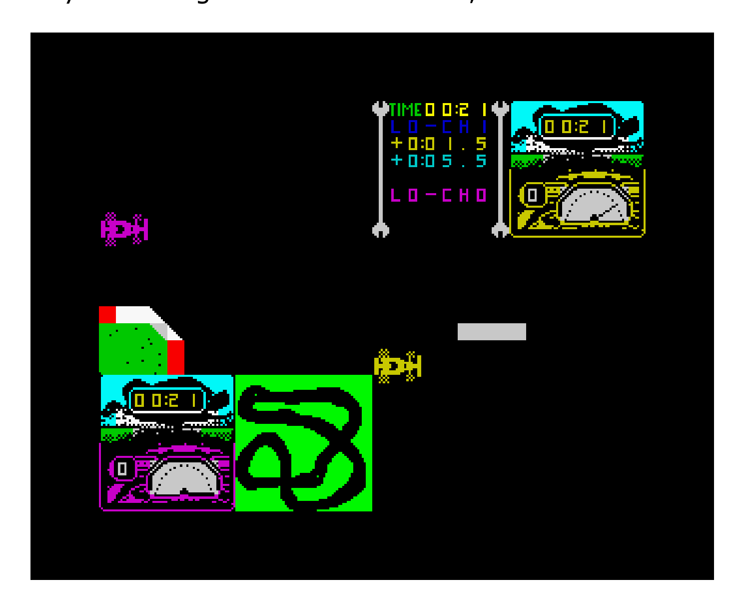
Los juegos de coches con un punto de vista cenital ya tienen un amplio historial en los Sinclair, aunque pocos nos vienen a la memoria por sus virtudes. Y menos, si además añadimos scroll a la ecuación. I need speed, inevitablemente nos recuerda el Aspar GP master de Dinamic, que aunque con motos, tiene un planteamiento similar: la disputa de un campeonato, en este caso contra tres adversarios, y con el añadido de poderlo hacer dos jugadores simultáneamente.

El programa en su inicio nos permite una personalización completa de la competición, junto con los diferentes niveles de dificultad de rigor. Tenemos ocho circuitos, unos cercanos a lo que podemos ver en nuestros televisores, y otros de factura ciertamente onírica. La música no deja de sonar, ni en el menú ni durante las carreras (en modelos 128), acompañada del ruido del motor. Aunque juguemos de forma individual, siempre tenemos en pantalla las acciones de dos coches. Hay teclas destinadas a cambiar la vista de los rivales, aunque, ¡cuidado! La zona del jugador uno también tiene activada esta función y podemos perder la vista de nuestro coche por una simple pulsación. Esta distribución de pantalla, sin separación entre ambas ventanas, genera confusión e incluso a la postre, algo de sensación de mareo. Nos encontramos con falsas continuidades de la pista, fruto de que nuestros ojos generan un efecto óptico de lo que en realidad no existe.




La inclusión de dos zonas de acción supone dedicar un espacio demasiado reducido a ambas, lo que nos va a provocar dos problemas más. El primero, que en calzadas demasiado amplias perderemos cualquier sensación de velocidad y de la ubicación espacial de nuestro coche. La segunda, que nuestras maniobras se verán afectadas ante los repentinos y a veces imposibles cambios de dirección, que nos va a tocar predecir a base de continuas consultas al mapa del circuito. Eso, si nos da tiempo para ello. Es el peaje a pagar por querer dar más con la fuerza de un z80.

El juego en su apartado gráfico se caracteriza por su simplicidad. Formulas uno de diseño más bien vintage acompañados por bordillos colchoneros, inmensas praderas de césped, y eremítica publicidad asfáltica de la editora. Todo ello empujado por un motor de scroll de carácter en carácter, que aunque muy meritorio, acentúa las dificultades de conducción de nuestro bólido. ¡Cualquiera diría que los caballos del motor prefieren la hierba a la gasolina! Los diferentes niveles de dificultad, al margen de las pistas, vienen sobre todo marcados por el comportamiento de los contrincantes, más afinado conforme el nivel que nos imponemos es mayor. Aunque hay que decir que quizás nuestro mayor enemigo es nuestro coche, cual Alonso dentro de un R-28.




No sé si I need speed es un proyecto demasiado ambicioso para las entrañas de un Spectrum, o si como dijo en su momento su autor el planteamiento inicial ha sido un lastre para el desarrollo del mismo, pero no termina de cuajar en términos de jugabilidad. Demasiados problemas en el control y la visibilidad penalizan la experiencia del usuario. Un proyecto que aunque finalizado y con aspectos técnicos interesantes, seguro que no dejará satisfecho del todo a sus autores. Esperemos a ver si a la segunda es la vencida.

Valoraciones

| | |
|---|---|
| Originalidad: | [4] |
| Gráficos: | [4] |
| Sonido: | [7] |
| Jugabilidad: | [4] |
| Adicción: | [3] |
| Dificultad: | [7] |

Javier Vispe

**Escuela de ladrones**

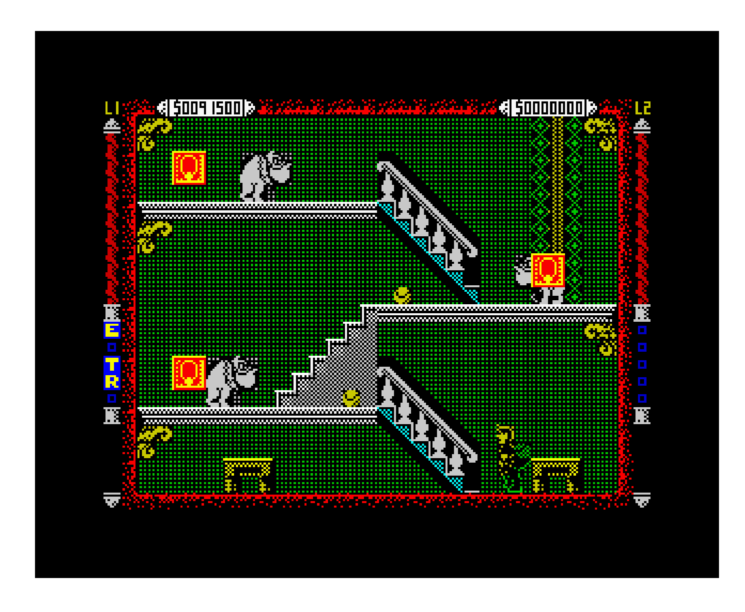



| | |
|---|---|
| Título | Escuela de ladrones |
| Género | Plataformas |
| Año | 2008 |
| Máquina | 128K |
| Jugadores | 1 ó 2 jugadores |
| Compañía | World XXI Soft Inc. |
| Autor | Ariel Ruiz, Alan Petrik |
| Otros comentarios | |

- Página oficial
- Compra la versión virtual (3,90€)
- Compra la versión física (8,75€)

¿Te imaginas que existiera una prestigiosa institución educativa con la misión de formar a los delincuentes del mañana? Esa es la premisa de la que parte Ariel Ruiz en su hasta ahora más ambicioso proyecto.

Dos jóvenes estudiantes, con poca fortuna en el examen final de la Escuela de Ladrones se ven abocados a hacer una recuperación de urgencia en un tour mundial de actos delictivos contra la propiedad privada. Todo sea por convertirse en personas de provecho el día de mañana. Armados de piedras, pelotas de tenis y poco más, van a hacerse los maestros del butrón, y el robo de guante no tan blanco. Enfrente, para impedirlo, perros guardianes, agentes de seguridad, arañas, momias, ratones que ni salidos de Caerbannog, robótica de última generación…



La propuesta de Escuela de ladrones se encuadra dentro del género de los juegos de plataformas en pantalla estática, donde el objetivo es acabar con todos los enemigos que circulan por el escenario y la recogida de objetos que aumenten nuestro marcador para pasar al siguiente nivel. Siguiendo la tradición que fuera decenas de veces imitada, tendremos también armas que nos faciliten el camino, y recogida de letras para obtener la palabra "EXTRA" con una buena recompensa de puntos. No sólo podremos contar con colaboración de un amigo, sino que también podremos optar por ver quien es el mejor de los dos en un duelo "a muerte".

En cada fase tendremos que hacernos rápidamente con objetos arrojadizos para ir debilitando a los enemigos hasta que caigan a la parte inferior de la pantalla, y con un golpe más mueran proporcionándonos armas, botiquines, objetos de valor o las ya citadas letras. La limpieza de fase sin recibir ningún golpe nos supondrá un bonus en la puntuación.

Brisa y Coraje son los dos personajes a elegir para la aventura. Cada uno con habilidades diferentes, nos darán ventaja en según que situaciones. Igualmente, los objetos arrojadizos que recogemos tienen una física distinta, a tener en cuenta para acertar en nuestra puntería al blanco. Los enemigos cuentan con comportamientos alternativos, aunque en ocasiones algunos dan sensación de excesiva aleatoriedad. De cuando en cuando, nos tocará enfrentarnos a alguno de grandes proporciones.

Todo esto, en el modo Acción, porque hay otra posibilidad llamada Sigilo. En ella el objetivo es de nuevo recoger objetos de valor y abrir cajas de caudales evitando ser detectados por los vigilantes. En ambos estilos contamos con una barra de energía, aunque en el último no vamos a poder regenerarla. Más difícil todavía, sobre todo a la hora de conseguir abrir las cajas fuertes.

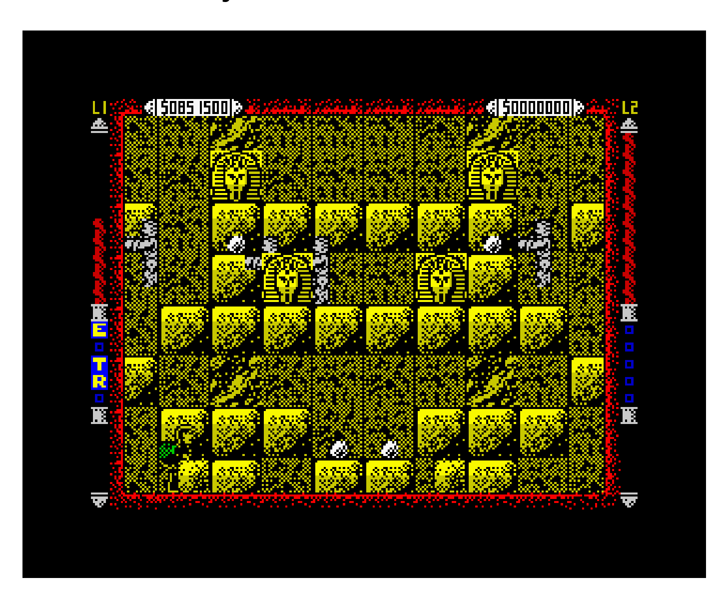


No hace falta que me extienda más en contar mecánicas para el jugador, tediosas en su descripción y lectura, cuando son esquemas lúdicos más que conocidos e incluso interiorizados. Posiblemente esa es al mayor virtud del juego, partir de una idea que funciona y darle pequeñas pinceladas que inviten al usuario a algo con aspecto parcialmente novedoso. Escuela de ladrones despliega un amplio abanico de personalizaciones y modos de juego que completan una oferta más que suficiente de cara a pasar horas ante su pantalla. Para los que se les haga corto, les quedará desbloquear los contenidos ocultos.

Sus más de sesenta niveles se ven precedidos de una presentación, bien surtida de imagen y sonido, aunque excesiva en el uso de digitalizaciones sin un tratamiento que las haga más amables. Una carga independiente, cosa de agradecer para cuando se trata de sólo llenar nuestro tiempo de asueto.

Nuestros avances en el examen nos llevan por escenarios muy diferentes y coloridos. Acabados con bastante acierto, en alguna ocasión para alguno pecarán de barroquismo, pero nos ambientan bastante bien en el lugar de acción. Cabe destacar algunos juegos de luces y sombras en las entrañas de la tierra. Lo mucho que se puede hacer con tan poco.

El motor gráfico del juego se ha realizado con SDV2, un entorno de desarrollo que permite trabajar a todo color, pero que a cambio ha sacrificado la precisión parcialmente. El movimiento se realiza carácter a carácter, restando fluidez a las animaciones. Los saltos y rebotes de los objetos quedan en parte deslucidos, y en alguna ocasión primeriza nos traicionarán. Este es el mayor inconveniente a marcar en el debe del juego, aunque no por ello decae el resultado en conjunto.



El apartado sonoro está muy bien resuelto por la mano del checo Alan Petrik (alias Factor6). Con gran experiencia en la demoscene de Amiga, Commodore, Spectrum y CPC, aquí nos sirve una ingente cantidad de melodías, desde covers de un cinematográfico Moby a tonadillas igualmente populares en el imaginario colectivo.

Escuela de ladrones raya a muy buen nivel a pesar del inconveniente del movimiento por caracteres. Aunque algunos modos de competición funcionan mejor que otros, es un juego para disfrutar, con multitud de opciones, acabado cuidado y elegante, e incluso con posibilidad de expansiones. Las horas de entretenimiento están garantizadas. Si sigue la progresión de Ariel, estamos de enhorabuena.

*Escuela de ladrones, también en cinta*

Compiler software inaugura una nueva etapa en la que también ofrece un servicio de edición en formato físico para aquellas personas que lo necesiten, realizando el grupo la inversión necesaria para que el juego llegue a

los usuarios de Spectrum. Escuela de ladrones ha sido el primer título que se acoge a esta iniciativa: cinta serigrafiada, instrucciones a todo color, en español e inglés, retractilado... Un servicio de altísima calidad y gran acabado que no dejará a nadie descontento. El juego se puede adquirir en cinta contactando con Compiler soft en su página web: http://compiler.speccy.org/




*Escuela de Ladrones versión física*

Valoraciones

Originalidad: [6]
Gráficos: [7]
Sonido: [8]
Jugabilidad: [7]
Adicción: [8]
Dificultad: [7]

Javier Vispe

**Nipik 2**

Título Nipik 2

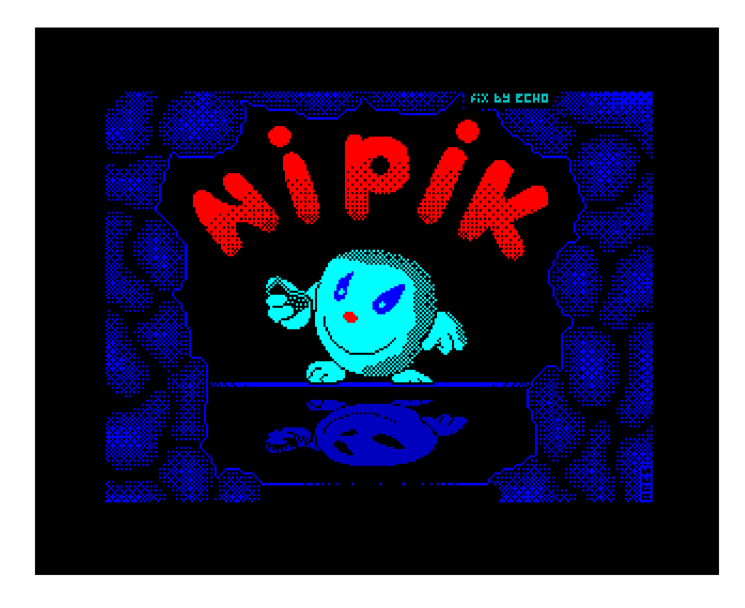
Portada Nipik 2

| | |
|---|---|
| Género | Arcade |
| Año | 2008 |
| Máquina | 128K |
| Jugadores | 1 jugador |
| Compañía | Triumph Game Labs. |
| Autor | Cj Echo, U-man, Karbofos, Transman, Kyv, Prof |

Otros comentarios

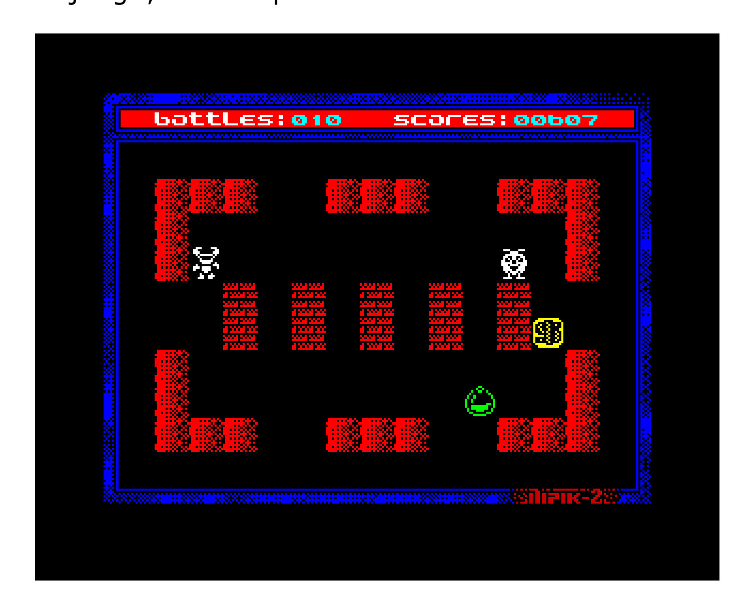
- Nipik 2 en WOS

Derivado de los distintos trabajos que llegaron a la tercera convocatoria del 'Your game' ruso, un breve (y leve) fogonazo de notoriedad alcanzó este Nipik 2 a pesar de no haber sido más que tercero entre las seis entradas por lo que refiere al volumen de votos finales (sabida es, por otra parte, la peculiar relevancia que este tipo de avales numéricos acostumbran a tener).



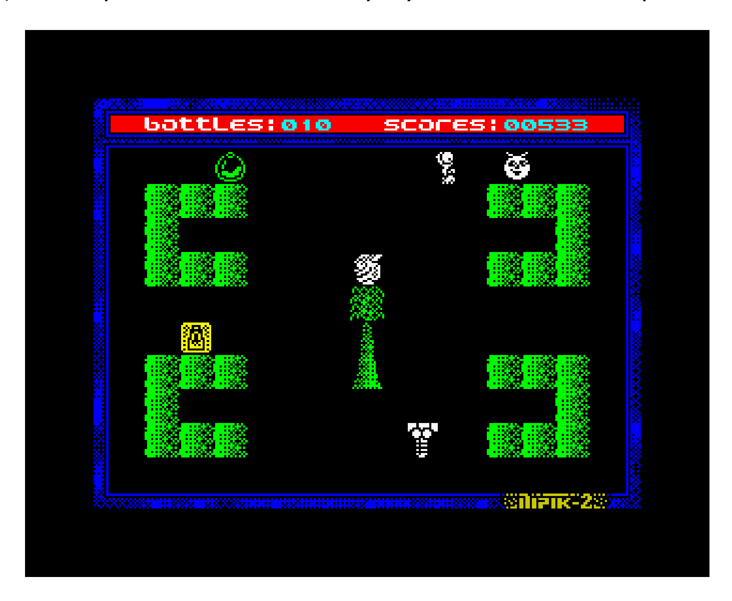

La trama no responde precisamente a nada que no hayamos podido observar ya en sencillos arcades añejos: sucesión de diversas pantallas a modo de laberinto en las que la aparición y posterior recogida de objetos (rechonchas botellas en el caso que nos ocupa) debe ir permitiendo la transición a niveles posteriores. Acompañando nuestro devenir se irán sumando hasta cinco clases de enemigos de distinta autonomía y peligrosidad; desde los absolutamente erráticos y poco dinámicos (en parte por culpa de algunas de las cerradas estructuras de pantalla) hasta los meramente perseguidores.

¿Power-ups? Como tales sólo serían calificables un par de los cuatro que pueden surgir en nuestra trayectoria (acelerador y aportador de vida extra). Otro de ellos aumentará nuestro score del momento (pasión desbordante la vivida en tal trance), mientras que el cuarto en discordia actúa en nuestra contra convirtiendo nuestro movimiento en pelín más flemático. Y caso de que todo ello (lo bueno y lo malo) nos venga más bien gordo a la hora de afrontar la partida de turno, siempre se pueden activar algunos de los cheats que se incluyen en el cargador previo al juego, incluído por defecto.



Nipik 2 consta de un total de 20 escenarios. La sinuosidad de caminos y el número de botellines a recoger en cada uno no sigue una ordenada heterogeneidad. Mismo baremo para la aparición de enemigos, algunos de los cuales acaban por ser, digámoslo de esta manera, esclavos de más de una esquina mal resuelta a nuestro favor. Hecho un balance conjunto de estos indicadores, puede afirmarse que solamente aparecen las más grandes y tediosas dificultades en los 3 ó 4 últimos pantallazos. Para el resto, y con menos o más paciencia y fortuna, trámite al canto.

Habrá que tener en cuenta las posibles (que desconozco con seguridad) condiciones de apuro bajo las cuales el juego recibió su desarrollo final de cara a poder ser presentado. Pero no exime ello de hablar de sus "taras" para con el jugador, siendo la que más resalta la naturaleza de movimiento del personaje bajo nuestro control, y esa inercia obligada (sea cual sea nuestra voluntad) que lleva a más de una pérdida de vida y a pensar en exceso en rutinas de previsión (haciéndose las rutas que escojamos mucho más complejas de llevar a cabo, sobre todo con el acelerador metido en nuestras carnes). Tampoco ayuda la especial morbidez de los enemigos de gama más baja, a los que les cuesta dios y ayuda salir de sus posiciones...




La vista no recoge apenas alegrías durante la partida. Excluyendo algunos motivos gráficos hacia el final del juego, los muros se basan apenas en dos (sólo dos) patrones gráficos distintos, bebiendo entre sí de la más evidente sosez al tiempo que quedándose en monocromía persistente. Y en lo referente al oído, aprobadín justo en cuanto a efectos, e in-game de los que sería mejor no haber incluído (cuando uno oye ese sexto grado menor acompañado de la tríada equivocada siente un chirrío de dientes que derrumba hasta la estética más mordaz de Walter Piston).

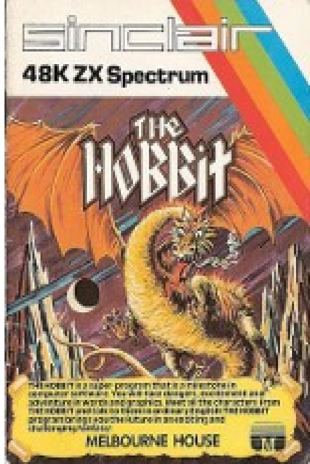
En conclusión: sirvámonos viajar una nimiedad en el tiempo (puestos a ser dramáticos, hasta ese iniciático 1982) y hagamos uso y disfrute de títulos como Bubble Trouble de Arcade Software. Nuestro metabolismo recreativo apenas sentirá cambio alguno, y más que seguramente paliaremos la variedad de enemigos que Nipik 2 ofrece con el adecuado grado de tensión que aquél sí supo aportar en su día.

| Valoraciones | |
|---|---|
| Originalidad: | [0] |
| Gráficos: | [3] |
| Sonido: | [3] |
| Jugabilidad: | [4] |
| Adicción: | [3] |
| Dificultad: | [6] |

Albert Valls

**The Hobbit**

| | |
|---|---|
| Título | The Hobbit |




| | |
|---|---|
| Género | Aventura Conversacional |
| Año | 1982 |
| Máquina | 48K |
| Jugadores | 1 jugador |
| Compañía | Melbourne House |
| Autor | Beam Software (Philip Mitchell, Veronika Megler) |

Otros comentarios

- The Hobbit en WOS
- Entrevista con Veronika Megler
- Página de Melbourne House (hoy Krome Studios)
- El Hobbit y sus errores, por Andrés Samudio, en Microhobby

Se ha hablado mucho de The Hobbit, siendo probablemente la aventura más famosa de todas las editadas para Spectrum, pero desde estas líneas de análisis, queremos hacerle un homenaje más, y tratar de acercaros a esta pequeña joya de la programación para Spectrum.

En 1980 se fundó Melbourne House, una pequeña compañía australiana que acabó por ser una de las grandes. Esta por aquel entonces pequeña compañía (como casi todas las creadoras de software de aquellos primeros años) llegaría a crear algunos de los juegos más afamados en la época dorada del Spectrum: The Way of the Exploding Fist, Penetrator, o el motivo de este artículo: The Hobbit, editado en 1982.



Estrictamente hablando, The Hobbit es una creación de Beam Software, cuyo nombre proviene de una contracción de las iniciales de sus creadores: Alfred Milgrom (experton en informática) y Naomi Besen (experta en marketing), siendo Melbourne House la distribuidora, aunque finalmente, siendo ambas compañías de Alfred y Naomi, acabarían por significar lo mismo.

Obviamente basada en la novela homónima de J.R.R. Tolkien, The Hobbit nos pone en el papel de Bilbo, un hobbit que emprende una trepidante aventura en compañía del mago Gandalf y el enano Thorin en pos del tesoro largo tiempo atrás robado por el malvado dragón Smaug.



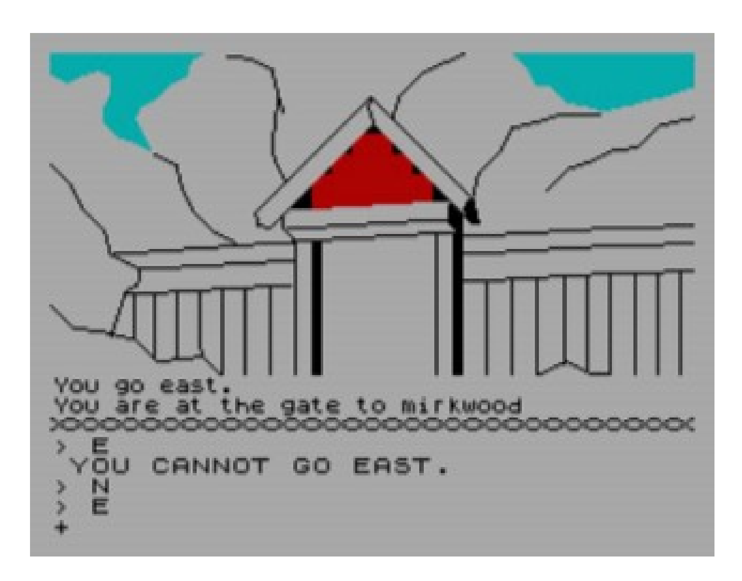

El juego fue inicialmente desarrollado para un TRS-80, pero cuando apareció el Spectrum se decidió continuar el desarrollo en la nueva máquina, en la cual salieron dos versiones, la original, y una segunda versión denominada 1.2, con varios errores corregidos. El desarrollo se llevó en total 18 meses.

Considerando la época el interprete de The Hobbit es muy avanzado, siendo capaz de entender frases complejas para otros parsers de la época, e incluso dar órdenes complejas a los compañeros del jugador (como Gandalf, Thorin, Elrond, y otros personajes del juego), si bien también es famoso por sus errores de programación, sobre los cuales hay incluso artículos. El juego admite ordenes en lo que los propios autores denominan INGLISH, una versión reducida del inglés, indicándose en las instrucciones que entiende hasta 500 palabras y puede realizar 50 acciones diferentes. Los mensajes emitidos por el parser, son en realidad plantillas con campos rellenables, de manera que el mismo mensaje podía reutilizarse decenas de veces simplemente cambiando el sujeto, el predicado, o el género y número.



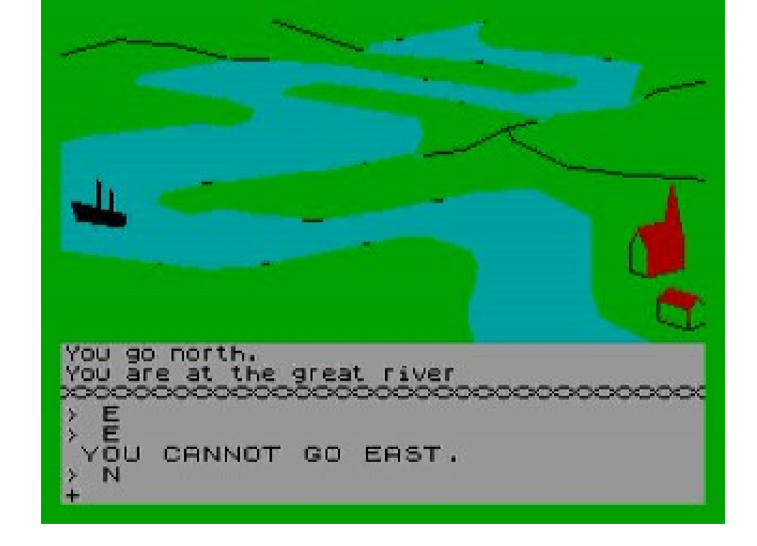

Alfred Milgrom quiso realizar el mejor juego de aventuras de toda la historia, y para ello contrató a Veronika Megler y Phillip Mitchell, ambos estudiantes de informática en la Universidad de Melbourne. Phillip fue el encargado de generar el intérprete, el engine del juego, mientras que Veronika se encargó de la codificación de lugares, objetos y personajes (llamados animals por los creadores), basándose en el propio libro de Tolkien en la medida de lo posible. Hoy en día es difícil contactar con Phillip, por lo que no se sabe mucho de él. Por su parte Veronika trabaja para IBM en EEUU. Ambos rondan la edad de 47 años. Otra gente colaboró en el proyecto, como el propio Alfred Milgrom y Stuart Ritchie, y los gráficos se basaron en los bocetos de de Kent Rees.

The Hobbit fue editado sin mucha expectativa por parte de Melbourne House: el portavoz de Melbourne decía en la Personal Computer World Show de 1983 "Esperabamos que el Hobbit muriera pronto, pero no hay signos de que eso vaya a ocurrir". Finalmente, fue un gran éxito de ventas de su época, llegando a alcanzar el millón de copias vendidas en todo el mundo (contando todas las plataformas). También fue probablemente un gran éxito en cuanto a copias piratas se refiere, basta con ver que aproximadamente dos de cada tres aventureros españoles que se iniciaron en aquella época lo hizo jugando al Hobbit, y que el Hobbit no se vendía en España. El éxito fue tal que se llegó a editar un libro de pistas que fue un éxito de ventas así mismo, escrito por David Elkan, y editado por Melbourne House Publishers. Dicho libro contiene la solución al juego en tres niveles, dando el primero breves pistas nada más, el segundo pistas y datos más claros, y el último, la solución orden por orden, de modo que el jugador pudiera avanzar sin romper el juego más que cuando fuera estrictamente necesario.




El éxito de The Hobbit llevó a Melbourne House a crear otros dos juegos continuación del mismo, que se basaron como no podía ser de otro modo en El Señor de los Anillos, el libro de Tolkien: Lord of the Rings y Shadows of Mordor, en los que se volvería a utilizar el motor creado por Mitchell, el cual sería también utilizado para otra famosa aventura de la casa: Sherlock. Así mismo, se editaron diversas parodias basadas en The Hobbit o en el resto de juegos de la saga como por ejemplo The Boggit o Bored of the Rings, ambos editados por Delta 4 y creados por Fergus McNeil.

En cuanto a la fidelidad con la obra de Tolkien, obviamente como todo juego se toma sus licencias, y así ciertos pasajes son ignorados mientras que otros requieren de nuestro ingenio. Podremos visitar el claro de los trolls, el bosque negro, la casa de Beorn o la Carroca, la ciudad de Lago, la montaña solitaria, Rivendell¿. y nos encontraremos con personajes del libro como Gandalf, Elrond, Thorin, Bardo o el propio dragón Smaug.




El juego engancha desde el principio, por la historia y por cómo se lleva ésta. Sin embargo ciertos detalles de aleatoriedad en los túneles de los goblins le hacen complicado a partir de cierto punto, lo cual hizo que muchos jugadores en su día no pasaran de ahí, y esta es la única pega que se le puede poner.

Desde el punto de vista actual el juego está ligeramente anticuado si lo comparamos con las aventuras conversacionales modernas, lo cual no es mucho decir teniendo en cuenta que ha cumplido 26 años, pero aún es probablemente una de las aventuras de Spectrum que merece la pena jugar si no la habéis jugado, y volver a jugar si ya lo hicistéis.

| Valoraciones | |
|---|---|
| Originalidad: | [6] |
| Gráficos: | [8] |
| Sonido: | [0] |
| Jugabilidad: | [8] |
| Adicción: | [9] |
| Dificultad: | [8] |

Carlos Sánchez

**Trilogía Farmer Jack**




| | |
|---|---|
| Título | Farmer Jack in Harvest Havoc, Farmer Jack and the hedge monkeys, Farmer Jack - Treasure trove |
| Género | Acción |
| Año | 2006-2008 |
| Máquina | 48K/128K |
| Jugadores | 1 jugador |
| Compañía | Cronosoft |
| Autor | Bob Smith, Lee du-Caine |
| Otros comentarios | |

- Web oficial

El más difícil todavía. A falta de hablar de un sólo juego, ¿qué tal lanzarse a por varios a la vez? Más concretamente, a por una trilogía que trata los "graves" problemas del "campo europeo". Todo esto, gracias al granjero Jack, que ya lleva tres aventuras de la mano de Bob Smith. Una saga que ha sido el comienzo de su colaboración con Lee Du-caine,y que a la postre, lo ha convertido en su músico fetiche. Tres juegos que son homenajes a arcades clásicos de principio de los años ochenta y que ha usado Bob como clínica de desintoxicación de otros trabajos.

¿Cuál fue la motivación de la idea? En su inicio el objetivo era crear un clon de Mr. Do, título con pobres adaptaciones al Spectrum. A partir de un argumento completamente surrealista surgió la figura del Jack: un granjero dispuesto a recoger su cosecha a pesar de correr un serio peligro a causa de derrumbamientos y extraños seres, quizás surgidos de mutaciones provocadas por las fumigaciones ilegales y los cultivos transgénicos. Había nacido "Farmer Jack in Harvest Havoc".



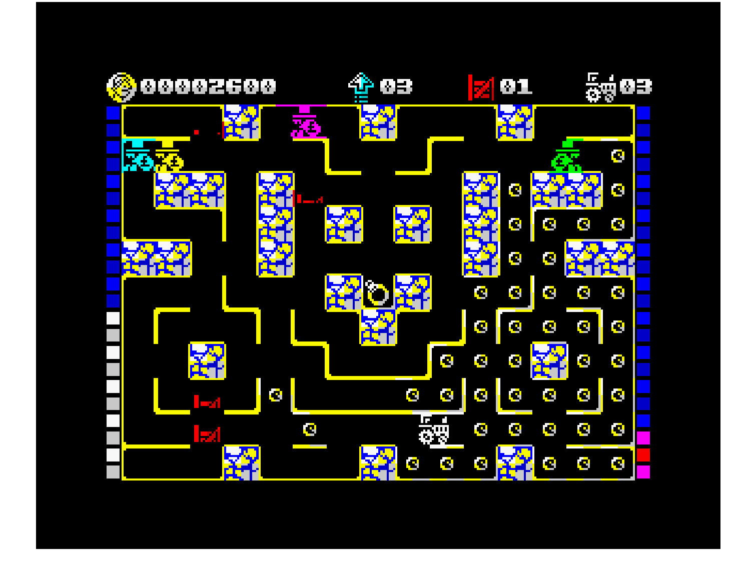

Enfrentados a persecuciones por angostos pasillos que ni salidas de la mano del señor King, y con un arsenal limitado de bombas como último recurso, estamos ante un heredero del género del laberinto, que tanto juego dio a principios de los ochenta. Sólo ante múltiples peligros, los cuales afectan a la fluidez del movimiento por exceso de carga al procesador. La arquitectura de los niveles también se ve afectada por ligeros halos de mezcla de atributos a nuestro alrededor, y de soluciones de color económicas a la hora de integrar la cosecha en los campos. A pesar de estos detalles, el granjero salió airoso de su primera cita con el Spectrum, pasando a formar parte del selecto club de personajes que vuelven a este ordenador.

A principios de 2008, Jack retorna con "The hedge monkeys", homenajeando otro clásico: Lady bug. En esta ocasión, de visita a su primo Jon, se encuentra con un gravísimo problema. Hordas de sucios hippies le invaden las tierras cada verano en busca del festival de Glastonbury arrasando sus campos. Que menos que ayudar a la familia... De nuevo otro escenario laberíntico, aunque sin tomarse licencias. El planteamiento respeta la idea original sin añadidos. Por ello, contamos con algunas paredes móviles, que nos sirven para huir de los urbanitas desbocados. Al margen de hacernos con la cosecha, contaremos con elementos de multiplicación de la puntuación y otros bonus, según la forma y el color de los mismos. Nada que sorprenda a nuestras canas, pero que sigue funcionando tan bien como el siglo pasado. Eso sí, aunque la mezcla de atributos sigue acompañando inevitablemente al tractor, su velocidad por fin es constante, y los decorados hacen gala de mayor mesura a la hora de combinar colores. La secuela supera a su antecesor de sobras.




La última parada hasta ahora del viaje de nuestro granjero es un destino ineludible. Si se trata de homenajear arcades clásicos de laberintos, no puede faltar el rey de reyes. "Farmer Jack Treasure Trove" no es lo que diríamos una reencarnación absoluta, pero en esencia es el enésimo retorno de Pac man con pinceladas de otro ilustre de las encerronas, Lock 'n' Chase.

Sin embargo, aquí no sólo nosotros nos afanamos en recoger las monedas romanas (¡vaya con la campiña inglesa y lo que encontramos en ella!). Los enemigos también han adoptado esta curiosa costumbre, y nos pueden tumbar en la partida superándonos en la cantidad de objetos conseguida. Por suerte, y literalmente, podremos poner puertas al campo. Efectivamente, tenemos la posibilidad de encerrar temporalmente a los falsos arqueólogos que nos roban el botín con vallas que además nos pueden sacar de alguna persecución peligrosa. Esta tercera entrega es la síntesis de un trabajo que ha ido yendo a más, puliendo defectos, experimentando sin perder el respeto por los orígenes. Aunque siguen evidentes los problemas de la gestión del color en el Spectrum por la propia estructura del juego, han sido elegantemente disimulados. La música, como siempre, no decepciona. Melodías alegres, con guiños a la música de pajar, y manteniendo un alto nivel, tanto en esta, como en las dos anteriores entregas. El señor Du-caine nos ha malacostumbrado.




No sabemos si Jack volverá a invadir las autopistas británicas con su tractor, con sus ideas conservadoras, tradicionales y de la vieja escuela. En otro caso, seguramente estarían fuera de lugar. En el Spectrum, todavía tienen vigencia.

| Valoraciones | |
|---|---|
| Originalidad: | [5] |
| Gráficos: | [6] |
| Sonido: | [8] |
| Jugabilidad: | [8] |
| Adicción: | [8] |
| Dificultad: | [7] |

Javier Vispe









# Save y Load: almacenamiento en cinta

¿Te has preguntado alguna vez en qué formato se almacenan los datos en una cinta de cassette para que el Spectrum pueda después cargar desde ellas pantallas de presentación, los personajes de un juego o el código ejecutable de un programa?

En este artículo vamos a mostraros cómo se graban y estructuran los datos en las cintas, y qué hace el Spectrum para acceder a ellos mediante las rutinas de que nos provee la ROM. Para aquellos de vosotros que queráis dar una aplicación práctica a lo que veremos hoy, se incluirá código de carga y grabación de pantallas y un ejemplo completo para probar.

**Formato de los datos en cinta**

Supongamos que, desde BASIC, salvamos un bloque de datos en cinta con el comando SAVE. Lo primero que nos interesa conocer es el *formato* o *estructura de los datos* almacenados en la cinta, es decir, ¿qué se guarda realmente en la cinta (en formato de audio) cuando hacemos un SAVE?

Un SAVE produce 2 bloques de datos en la cinta:

- Un bloque de 19 bytes de tamaño fijo, conocido como cabecera. Este bloque es cargado muy rápidamente, debido a su pequeño tamaño. Si pensáis en los cientos ó miles de LOADs desde cinta que habréis hecho en vuestro Spectrum, recordaréis, nada más pulsar PLAY, la aparición del tono guía (líneas del borde rojas y cyan) seguido de un brevísimo momento de carga de datos (líneas del borde azules y amarillas). Es en ese momento en el que aparece en pantalla la información relativa al juego/programa que estamos cargando.
- Un bloque de longitud variable que contiene los datos concretos y reales a cargar.

Ambos bloques son en realidad "datos" con el siguiente formato:

- Un byte inicial, que como veremos se llama Flag Byte.
- Los datos en sí mismos: 17 bytes para cabeceras, o la longitud concreta de los datos para los bloques de datos.
- Un byte de checksum o CRC.

Profundicemos un poco más en estos 2 bloques de datos:

Cada bloque se inicia con una serie de pulsos de 2168 t-stados cada uno, que constituyen el tono guía. La cantidad de pulsos (la duración) de este tono guía es de 8063 pulsos para los bloques de cabecera, y 3223 pulsos para los bloques de datos. Es por eso que la duración del tono guía (el famoso pitido inicial de la carga) es mayor para la carga de la cabecera que para el de los datos en sí mismos. Es decir, el tono guía está presente tanto para los bloques de cabecera como para los de datos, salvo que su duración es menor en los bloques de datos.

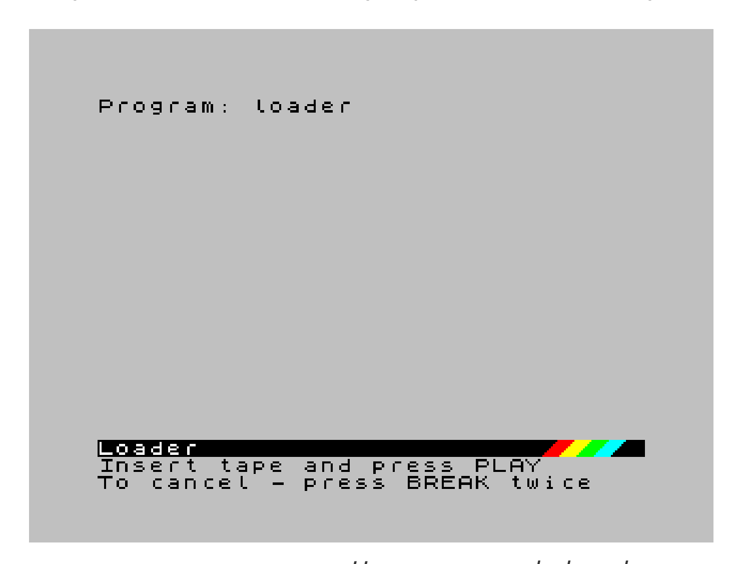
Un **t-stado** (t-state) es 1 ciclo de reloj, y equivale a 1 / 3.500.000 segundos.

Veamos una figura donde se muestra el estado del **mapa de memoria del Spectrum**:




*Aspecto del tono guía*

Tras el tono guía de la cabecera viene la cabecera en sí misma (que no dejan de ser datos). Su carga, como ya hemos dicho, tarda un tiempo muy corto en realizarse, ya que son sólo 19 bytes a ser leídos desde el cassette.




*Una vez cargada la cabecera.*

En la imagen anterior podemos ver el aspecto de la pantalla una vez terminado el tono guía y cargada la cabecera. Esta cabecera, mediante sus 19 bytes, le indican al Spectrum la naturaleza de los datos a cargar en el bloque de datos que sigue a la misma, con el siguiente formato:

| Byte | Longitud | Descripción |
|---|---|---|
| 0 | 1 | Byte Flag (0x00 ó 0xFFF). |
| 1 | 1 | Tipo de bloque (0-3). |
| 2 | 10 | Nombre de fichero (rellenado con espacios en blanco). |
| 12 | 2 | Longitud del bloque de datos a cargar. |
| 14 | 2 | Parámetro 1. |
| 16 | 2 | Parámetro 2. |
| 18 | 1 | Checksum / Suma de comprobación. |

Pasemos a describir los diferentes campos de la cabecera:

El byte de flag (Byte 0) y el de Checksum (byte 18) no forman parte exactamente de la cabecera, sino del bloque cargado en sí mismo (también están presente cuando cargamos datos y no cabeceras), pero se han incluído dentro de la tabla para hacerla de lectura más sencilla. Puede decirse que el byte-flag es el byte prefijo de todo bloque de datos (considerando una cabecera de 17 bytes como un bloque de datos) y el checksum es el byte sufijo de ese mismo bloque. Concretamente, el valor de Byte-Flag es de 0x00 para bloques de cabecera y 0xFF para bloques de datos.

El byte de tipo de bloque indica qué datos se van a cargar a continuación, según los siguientes valores:

| Valor | Significado |
|---|---|
| 0 | Programa. |

1 Array de números.

2 Array de caracteres.

3 CODE (datos a cargar en memoria).

En el caso de bloques de tipo CODE, el byte "Parámetro 1" define la dirección de inicio del bloque de código cuando se hizo el SAVE, y el parámetro 2 contiene el valor 32768.

Para bloques de tipo PROGRAM, el Parámetro 1 contiene el valor de la línea BASIC de autostart (o un número mayor de 32768 si no se dio un parámetro LINE al hacer el SAVE), y el Parámetro 2 contiene el inicio del área de variables relativa al inicio del programa.

Un detalle: una pantalla de datos (SCREEN$) se define en esta cabecera como un bloque de tipo CODE de 6912 bytes a cargar sobre la dirección 16384.

Tras la cabecera (tono guía + bloque datos) viene el bloque de datos en sí mismo, que vuelve a componerse de un tono guía, el Flag Byte, los datos y el checksum.



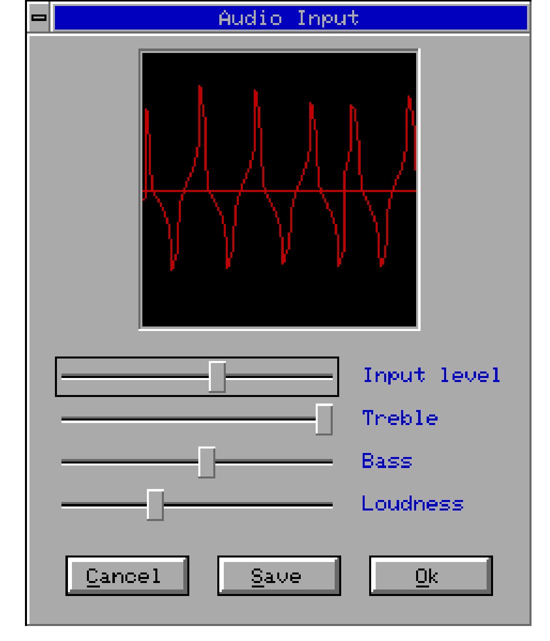

*Aspecto de la carga del bloque de datos en sí mismo.*

Ahora bien ... ¿cómo es posible que se almacenen como audio datos digitales? ¿Cómo entiende el Spectrum si los datos que está cargando son unos o ceros y los agrupa en bloques de bytes que podemos interpretar de forma lógica?

Como ya hemos comentado al hablar del tono guía, la clave está en la temporización precisa a la hora de leer datos desde la cinta. Aparte de tonos guía y pulsos de sincronización ... ¿qué es un cero en la cinta? ¿qué es un uno? ¿Cómo se almacena (y lee) un byte de 8 bits? ¿Y cómo se almacena (y lee) un conjunto de bytes?

- Un cero (bit=0) se codifica en cinta como 2 pulsos de una duración de 855 t-stados cada uno.
- Un uno (bit=1) se codifica en cinta como 2 pulsos de una duración de 1710 t-stados cada uno.
- Para almacenar los 8 bits de un byte, se almacenan bit a bit de mayor a menor peso (primero el bit 7, luego el 6, el 5, el 4, el 3, el 2, el 1 y finalmente el LSB o bit 0).
- Cuando se almacena más de un byte (un bloque) se guardan primero los datos del primer byte del bloque, luego el segundo, etc.

Los pulsos son ondas con un aspecto como el siguiente (aspecto de un tono guía en TAPER):




*Aspecto de un tono guía (TAPER).*

Es decir, si tenemos que almacenar en cinta los siguientes bytes:

```
abcdefgh ijklmnop
```

Se almacenarían en cinta en este orden:

```
a b c d e f g h i j k l m n o p
```

Así pues, la rutina de la ROM del Spectrum se encarga (tanto al grabar como al leer) de codificar pulsos de diferentes duraciones para almacenar los ceros y unos de forma consecutiva. Nosotros aprovecharemos (como veremos a continuación) dicha rutina para cargar o salvar bloques de datos a nuestro antojo sin tener que programar esas temporizaciones y lecturas/escrituras a la cinta. Para nosotros será tan sencillo como cargar los valores adecuados en ciertos registros y realizar un CALL. No obstante, para los más curiosos, al final de este capítulo tenéis un enlace a las rutinas de la ROM adecuadamente comentadas de "The Complete Spectrum ROM Disassembly", por el Ian Logan y Frank O'Hara (publicado en 1983).

El nivel más bajo al que necesitamos llegar es el siguiente:

- Cada bloque tiene la siguiente estructura física:
    - Un tono guía de 8063 (cabeceras) ó 3223 pulsos (datos) de 2168 t-stados cada uno.
    - Un pulso de sincronización de 667 t-stados.
    - Un segundo pulso de sincronización de 735 t-stados.
    - El bloque de datos en sí mismo, bit a bit (0 = 2 pulsos de 855 t-stados, 1 = 2 pulsos de 1710 t-stados).
- Cada bloque tiene la siguiente estructura lógica (formato de los datos DENTRO de un bloque):
    - Flag byte, con un valor de 0x00 para bloques de cabecera o 0xFF para bloques de datos.
    - Los datos en sí mismos: 17 bytes para cabeceras, o la longitud concreta de los datos para los bloques de datos.
    - Un byte de checksum, calculado de forma que haciendo un XOR de todos los bytes juntos, incluyendo el flag Byte, produzca 0x00.

Resumiendo:

- Los datos salvados en cinta constan de tono guía, seguido de un bloque de datos de 19 bytes denominado cabecera, seguido de un tono guía de menor duración que el inicial (por ser de datos), seguido de los datos en sí mismos.
- Esa cabecera proporciona información sobre el nombre, duración y ciertos parámetros relativos a los datos en sí mismos (el segundo bloque cargado).
- Los datos se leen de cinta secuencialmente, del primer al último byte del bloque de datos, y almacenado cada byte desde el bit 7 al 0 secuencialmente.
- Los 1s y los 0s se almacenan en cinta como pulsos de duraciones concretas.
- Las rutinas de la ROM nos permiten leer y escribir en cinta bloques de datos, realizando ellas la temporización adecuada para convertir nuestros "datos en memoria" en "pulsos" o viceversa sin complicación por nuestra parte.
- Sólo necesitaríamos escribir una rutina propia de carga (que detecte pulsos, temporice, etc) si

quisiéramos programar nuestra propia carga, por ejemplo para cargar a diferente velocidad que el Spectrum (ultracargas), que cargue de dispositivos externos (puerto de joystick, algún pin concreto del bus de expansión, etc.) o para ejecutar código mientras cargamos el programa (minijuegos durante la carga, contadores de carga, etc). Para el resto de casos, bastará con usar mediante CALL las rutinas de la ROM.

**Ejemplo de volcado de un SAVE**

En la FAQ de comp.sys.sinclair y WorldOfSpectrum tenemos un ejemplo muy interesante que muestra el formato lógico de los datos grabados con un SAVE en cinta. Supongamos el siguiente comando BASIC:

```
SAVE "ROM" CODE 0,2
```

Este comando BASIC salvaría (SAVE), un total de 2 bytes (2) de datos (CODE), empezando en 0 (0) a cinta. En resumen, salvaría el contenido de la dirección de memoria 0x0000 y 0x0001 en cinta. Esto produciría los siguientes datos en cinta:

```
00 03 52 4f 4d 20 20 20 20 20 20 20 02 00 00 00 00 80 f1 ff f3 af a3
```

Comencemos mostrando qué representa cada dato poco a poco:

| **Cabecera** | **Bloque de datos** |
|---|---|
| 00 03 52 4f 4d 20 20 20 20 20 20 20 02 00 00 00 00 80 f1 | ff f3 af a3 |

Desgranando más la información:

| **Byte Flag** | **Datos cabecera** | **Checksum** | **Byte flag** | **Datos ROM** | **Checksum** |
|---|---|---|---|---|---|
| 00 | 03 52 4f 4d 20 20 20 20 20 20 20 02 00 00 00 00 80 | f1 | ff | f3 af | a3 |

Concretamente, los datos de la cabecera (ignorando el Byte Flag y el Checksum):

| **Tipo de bloque** | **Nombre de fichero** | **Longitud bloque** | **Parámetro 1** | **Parámetro 2** |
|---|---|---|---|---|
| 03 | 52 4f 4d 20 20 20 20 20 20 20 | 02 00 | 00 00 | 00 80 |

Como véis el nombre "**ROM**" (52 4F 4D) se completa con espacios en blanco hasta los 10 caracteres. Además podemos ver la longitud del bloque que se salvó (02 bytes).

Después de estos datos tenemos el checksum (F1) y el bloque de datos en sí mismo:

| **Byte Flag** | **Datos grabados** | **Checksum** |
|---|---|---|
| ff | f3 af | a3 |

En este caso el byte flag es 0xFF (bloque de tipo "datos"), al cual siguen los 2 bytes tomados de la ROM y grabados a cinta (0x0000 y 0x0001) y el checksum (0xA3).

**Rutinas de carga de la ROM**

Ya sabemos cómo se almacenan los datos en cinta, así que nuestra próxima misión es conocer cómo cargarlos o grabarlos de una manera sencilla. Para hacer esto usaremos las funciones de la ROM del Spectrum para carga y grabación de datos a cinta: hablamos de 2 subrutinas (de LOAD y SAVE) a las que podremos llamar con unos parámetros concretos.

**Rutina de LOAD de la ROM**

La rutina de LOAD comienza en la dirección $0556 (0556h ó 1366d) y requiere los siguientes parámetros:

| **Registro** | **Valor** |
|---|---|
| IX | Dirección inicio de memoria donde almacenar los datos que se van a cargar. |
| DE | Longitud del bloque de datos a cargar. |
| A | Flag Byte, normalmente 0x00 para cargar cabeceras o 0xFF (255) para cargar datos. |

CF (CarryFlag) 1=LOAD, 0=VERIFY

La rutina devuelve el CF = 0 si ocurre alguno de los siguientes errores:

- "R-Tape Loading Error" (bien por timeout o bien por byte de paridad incorrecto).
- Flag Byte incorrecto.
- "D BREAK - CONT repeats" (se pulsó la tecla BREAK).

Recuerda que puedes activar el CARRY FLAG con la instruccion "SCF" (Set Carry Flag) y ponerlo a cero con un simple "AND A".

Veamos 2 ejemplos, el primero cargaría una pantalla gráfica sobre la videomemoria (el equivalente de un LOAD "" SCREEN$) siempre y cuando la pantalla se haya grabado sin cabecera:

```
SCF                 ; Set Carry Flag -> CF=1 -> LOAD
LD A, 255           ; A = 0xFF (cargar datos)
LD IX, 16384        ; Destino del load = 16384
LD DE, 6912         ; Tamaño a cargar = 6912
CALL 1366           ; Llamamos a la rutina de carga
```

Este segundo programa cargaría un bloque de código ejecutable en memoria, y saltaría a él (un programa "cargador"):

```
SCF                ; Set Carry Flag (LOAD)
LD A, 255         ; A = 0xFF (cargar datos)
LD IX, 32768       ; Destino de la carga
LD DE, 12000       ; Nuestro "programa" ocupa 12000 bytes.
CALL 0556         ; Recordemos que 0556h = 1366d
JP 32768           ; Saltamos al programa código máquina cargado
```

**Rutina de SAVE de la ROM**

La rutina SAVE de la ROM tiene unos parámetros muy similares a la de LOAD, y está alojada en 1218d (04c2h):

| Registro | Valor |
|---|---|
| IX | Dirección inicio de memoria de los datos que se van a grabar. |
| DE | Longitud del bloque de datos a grabar (se grabarán los datos desde IX a IX+DE). |
| A | Flag Byte, 0x00 para grabar cabeceras o 0xFF (255) para grabar datos. |
| CF (CarryFlag) | 0 (SAVE) |

Lo normal es que no tengamos que recurrir en prácticamente ninguna ocasión a la rutina de grabación de datos, de modo que nos centraremos, mediante ejemplos, en la rutina de carga.

**Cargando o Ignorando la cabecera**

Cuando salvamos datos desde BASIC, lo normal es que se generen 2 bloques de datos, el de la cabecera, y el de los datos en sí mismos. El bloque de datos de la cabecera, cargado en memoria, nos permite saber el tamaño y destino de los datos que vendrán en el siguiente bloque. Es decir, cargando el primer bloque obtenemos la información necesaria para cargar en IX y DE los valores adecuados para la carga del bloque de datos.

En ocasiones, podemos ignorar el bloque de cabecera totalmente, sobre todo cuando sabemos qué vamos a cargar desde cinta, qué destino tiene, y qué tamaño tiene, y lo especificamos directamente en nuestro programa ASM. En ese caso, podemos cargar la cabecera con el CARRY FLAG a cero (verify), con lo cual la leemos pero no la almacenamos en memoria, y después cargar los valores adecuados en IX, DE, A, etc, poner el CF a 1, y cargar los datos que vienen tras la cabecera.

Supongamos que grabamos un bloque de datos, gráficos, una pantalla o música en cinta usando SAVE, rutinas de la ROM, o desde un emulador o herramienta cruzada de PC. Supongamos que sabemos el tamaño exacto en cinta de dichos datos, y no necesitamos leer y analizar la cabecera para cargarlos. En tal caso, podemos ejecutar código como el siguiente:

```
AND A                         ; CF = 0 (verify)
CALL 1366                     ; Cargamos e ignoramos la cabecera

SCF                           ; Set Carry Flag -> CF=1 -> LOAD
```

```
LD A, 255                    ; A = 0xFF (cargar datos)
LD IX, direccion_destino     ; Destino del load
LD DE, tamaño_a_cargar       ; Tamaño a cargar
CALL 1366                    ; Llamamos a la rutina de carga
```

Posteriormente veremos un ejemplo que ignora la cabecera al cargar una pantalla SCR completa sobre la videoram.

¿Cuándo nos interesa analizar la cabecera? Principalmente cuando los datos están generados desde nuestro propio programa y tienen un tamaño variable. Por ejemplo, supongamos que programamos un editor de textos que da al usuario la oportunidad de salvar y cargar los textos en cinta. En tal caso, necesitaremos leer la cabecera para saber el tamaño del documento (bloque de datos) a cargar, ya que no lo conocemos de antemano.

No obstante, en el caso de un juego, normalmente se conoce con antelación el tamaño de los datos a cargar, por lo que se puede ignorar felizmente la cabecera del bloque de cinta.

**Convirtiendo datos en cinta**

Lo primero que necesitamos saber es, ¿cómo convertimos nuestros datos (gráficos, pantalla de carga, números, tablas precalculadas, sprites, música, etc) en datos cargables desde las rutinas que hemos visto? Hay múltiples formas de hacerlo.

Para empezar, podemos hacerlo desde el mismo BASIC del Spectrum, usando el comando SAVE: esto nos permitirá grabar datos de memoria en cinta:

```
SAVE "nombre" CODE direccion_inicio, tamaño
```

En la mayoría de los casos, muchos de nosotros programamos hoy en sistemas PC usando compiladores cruzados, ensambladores cruzados y emuladores, por lo que normalmente lo que nos interesará es obtener ficheros TAP para poder concatenarlos con nuestros cargadores o programas.

Supongamos que estamos programando un juego que, nada más acabar, lo primero que hace es cargar desde cinta los datos del nivel actual (gráficos, mapeado, etc). Esto implica que cuando programemos el juego, tendremos por un lado el código, que nos proporcionará un fichero TAP (por ejemplo) listo para ejecutar. A ese fichero TAP tendremos que concatenarle los datos de los diferentes niveles (o gráficos, o los datos que necesitemos).

Así, nuestro "programa.asm" (código fuente) se convierte en "programa.bin" tras el proceso de ensamblado, y finalmente obtenemos un "programa.tap" (o .tzx) listo para cargar en un Spectrum.

Pero en dicho TAP o TZX tenemos que añadir (al final del mismo) los datos que el programa espera cargar. Imaginemos que estos datos son una pantalla gráfica (.scr) de 6912 bytes. Tendremos un fichero "pantalla.scr", y tenemos que introducirlo dentro de nuestro fichero TAP, al final, tras el código del programa, para que cuando este sea ejecutado, lo siguiente que cargue desde cinta nuestro programa sea dicha pantalla SCR.

Para ello, con el objetivo de hacerlo de una manera muy sencilla, utilizaremos ficheros TAP. El formato de este tipo de ficheros es muy sencillo, simplemente contienen bloques de datos precedidos por 2 bytes que indican el tamaño del bloque. Supongamos que tenemos en un fichero los 2 primeros bytes de la ROM que vimos anteriormente:

```
f3 af
```

Este fichero de 2 bytes de tamaño (inicio_rom.bin, por ejemplo), guardado en una cinta tendría el formato que vimos anteriormente: 2 bloques (cabecera y datos).

```
00 03 52 4f 4d 20 20 20 20 20 20 20 02 00 00 00 00 80 f1

+

ff f3 af a3
```

Es decir, 2 bloques de cinta de 19 y 4 bytes de datos, que conforman un SAVE de nuestros 2 bytes. Pues un fichero TAP con estos datos sería, sencillamente, el escribir estos 2 bloques en un fichero anteponiendo a cada bloque el tamaño a cargar:

```
13 00 00 03 52 4f 4d 20 20 20 20 20 20 20 02 00 00 00 00 80 f1

+

04 00 ff f3 af a3
```

Es decir "13 00" (número 19 en formato WORD, indicando el tamaño del bloque que viene a continuación) seguido de los 19 bytes, y "04 00" (número 4 en formato WORD) seguido de los 4 bytes del bloque.

El contenido en binario de nuestro inicio_rom.tap sería, pues:

```
13 00 00 03 52 4f 4d 20 20 20 20 20 20 20 02 00 00 00 00 80 f1 04 00 ff f3 af a3
```

Y el tamaño resultante en bytes del fichero serían 2 + 19 + 2 + 4= 27 bytes.

Gracias a este formato tan sencillo, podemos unir ficheros TAP simplemente concatenándolos. De esta forma, si tenemos nuestro "programa.tap" y la "graficos.tap", y queremos unirlos porque nuestro programa, al ejecutarse, carga los gráficos esperándolos en cinta tras el código del mismo, bastaría con hacer:

```
Linux:    cat programa.tap graficos.tap > programa_final.tap
Windows:  copy /B programa.tap graficos.tap programa_final.tap
```

Sabemos cómo podemos obtener nuestro programa en formato TAP: cogemos el código fuente, lo compilamos, y o bien obtenemos un TAP directamente (pasmo --tapbas), o bien obtenemos un BIN que convertimos con BIN2TAP. Pero ... ¿cómo convertimos nuestro "graficos.bin", "pantalla_carga.bin", "musica.bin" o cualquier otro fichero de datos en crudo? No podemos usar el BIN2TAP original porque éste añade un cargador BASIC al principio del programa... hay múltiples soluciones, pero la más sencilla es utilizar un ensamblador como **pasmo**.

Para convertir un fichero .bin en un fichero tap sin cabecera, creamos un pequeño programa ASM (rom.asm) como el siguiente:

```
INCBIN "rom.bin"
```

A continuación, "ensamblamos" ese programa con PASMO indicando que queremos que nos genere un TAP sin cabecera BASIC:

```
pasmo --tap rom.asm rom.tap
```

Con esto, obtendremos un fichero "rom.tap" con el contenido de rom.bin, y sin cargador BASIC, listo para utilizar.

**Ejemplo completo**

Finamente, para aquellos programadores que quieran ver un ejemplo de aplicación práctica de la recuperación de datos desde un binario en ejecución, vamos a juntar todo lo visto para realizar un programa que cargue una pantalla gráfica completa (fichero .scr) sobre la videoRAM.

Los pasos a seguir para generar el ejemplo son los siguientes:

Primero, buscamos un fichero .SCR de carga (por ejemplo, la pantalla de carga de cualquier juego obtenida desde WOS InfoSeek) y lo almacenamos en disco.

Segundo, mediante pasmo obtenemos un TAP con los datos del fichero SCR, sin cabecera BASIC. Dicho TAP tendrá un tamaño como el siguiente:

```
[sromero@compiler]$ ls -l zxcolumns.*
-rw-r--r-- 1 sromero sromero 6912 2007-10-08 13:01 zxcolumns.scr
-rw-r--r-- 1 sromero sromero 6937 2007-10-08 13:02 zxcolumns.tap
```

Ahora ya tenemos una pantalla SCR guardada en formato TAP (en cinta). Nótese cómo podríamos cargar este TAP desde BASIC con un *LOAD "" CODE 16384,6912*, y aparecería la imagen en pantalla.

Lo siguiente que necesitamos es el programa propiamente dicho, el cual hará la carga de la pantalla en videomemoria:

```
;-----------------------------------------------------------------------
; Loadscr.asm : Demostración de las rutinas LOAD de la ROM, con
; la carga de un fichero SCR (desde cinta) en videomemoria.
;-----------------------------------------------------------------------

  ORG 32000

  AND A                              ; CF = 0 (verify)
  CALL 1366                          ; Cargamos e ignoramos la cabecera

  SCF                                ; Set Carry Flag -> CF=1 -> LOAD
  LD A, 255                          ; A = 0xFF (cargar datos)
  LD IX, 16384                       ; Destino del load = 16384
  LD DE, 6912                        ; Tamaño a cargar = 6912
  CALL 1366                          ; Llamamos a la rutina de carga

  RET

END 32000
```

Al respecto del código fuente, como habréis notado, realizamos 2 llamadas a la rutina de la ROM. La primera carga (pero no almacena en ningún sitio) el primer bloque de datos existente (la cabecera de la pantalla de carga). La rutina de la ROM ignorará esta carga porque el CARRY FLAG está a cero (0=VERIFY). La segunda llamada a 1366 realizará la carga de los datos propiamente dichos. Al cargarlos sobre la dirección de destino 16384 (la dirección de la videoram), veremos cómo se van cargando sobre la pantalla directamente desde la cinta.

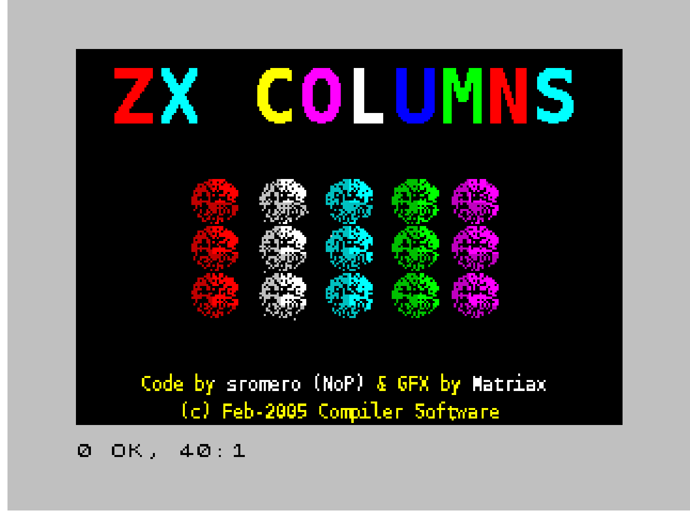
Ensamblamos nuestro programa con "pasmo --tapbas loadscr.asm loadscr.tap" y tendremos lo siguiente:

- Un TAP (loadscr.tap) con nuestro programa (pero que no tiene datos después de él).
- Un TAP (zxcolumns.tap) con los datos gráficos (en este caso, una pantalla completa de 6912 bytes).

Si cargamos en el Spectrum, o en un emulador, el fichero loadscr.tap, nos encontraríamos con que intenta cargar los datos desde cinta, pero no hay datos almacenados tras el programa. Para solucionarlo, concatenamos los 2 TAPs (con cat en Linux o copy en Windows/DOS):

```
[sromero@compiler]$ cat loadscr.tap zxcolumns.tap > programa.tap
```

Ahora sí, cargando "programa.tap" en el emulador, cargaremos nuestro programa, el cual llama a la rutina de la ROM para cargar los datos que van después del programa (la pantalla de carga) en videomemoria. Si lo probáis en un emulador, recordad deshabilitar las opciones de carga rápida o carga instantánea si queréis ver el efecto de la carga.




*Nuestro programa cargando el fichero SCR.*

Si os fijáis durante la carga, veréis como primero se carga el LOADER, luego el código máquina de nuestro programa, y después la pantalla. Contando los tonos guía de carga también encontraréis el lugar donde se lee, pero ignora, la cabecera (19 bytes, carga muy corta) de la pantalla SCR.

Un apunte: tanto en el caso de la carga, como de la grabación de datos, recordad que las rutinas de la ROM no indican al usuario que debe pulsar PLAY o REC, por lo que debemos indicar al usuario cuándo debe pulsar PLAY o REC dentro de nuestros programas o juegos. Incluso, cuando acabemos de cargar los datos relativos a

nuestro juego, resulta conveniente indicarle al usuario cuándo debe detener la cinta (especialmente en juegos multicarga) e insertar los segundos de "espacio" que creamos convenientes entre bloques.

Recordad que en este ejemplo hemos cargado 6912 bytes de un fichero SCR directamente sobre la videoRAM, pero nada nos impide cargar ficheros de cualquier otro tamaño, con cualquier otro contenido (sprites, fondos, datos, mapeados) sobre cualquier lugar de la memoria (asegurándonos primero que no hay nada en el lugar destino de la carga, como código, la pila, u otros datos o variables).

Así pues, de la misma forma que hemos cargado una pantalla SCR, podemos organizar los gráficos y mapeados de nuestro juego en 1 "bloque de datos" por nivel, cargar los datos del nivel 1 tras acabar la carga de nuestro programa, y sobreescribir estos gráficos y mapeados "en memoria" con los de los diferentes niveles según vaya avanzando el jugador. En otras palabras, podemos hacer un juego multicarga que nos permita tener más sprites, pantallas o gráficos disponibles para cada nivel, que los que tendríamos disponibles si cargamos todos los datos de todos los niveles del juego, ya que usamos toda la memoria para cada nivel, en lugar de dividirla en espacio para los diferentes niveles. A cambio, el usuario tendrá que cargar desde cinta las diferentes fases conforme avanza, y rebobinar para cargar los datos del "Nivel 1" cuando deba empezar una nueva partida.

**Ficheros**

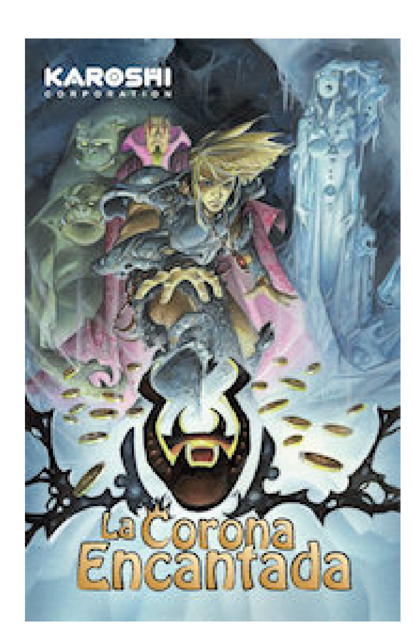
- Ejemplo de load de SCR sobre la VRAM.
- TAP del ejemplo anterior.
- Conversor de datos a fichero TAP.
- Bin2tap que no genera cargador BASIC.

**Enlaces**

- FAQ 48K de WOS.
- Tape Data Structure.
- Cintas y TAPER.
- Rutina de carga/grabación de la ROM, desemsamblada y comentada.
- Manual del +3.
- Web del Z80.
- Z80 Reference de WOS.
- PASMO.
- BIN2CODE (Web de Metalbrain).

Santiago Romero

# LA CORONA ENCANTADA




| | |
|---|---|
| Título | La Corona Encantada |
| Género | Plataformas |
| Año | 2009 |
| Máquina | ZX Spectrum 48K/128K |
| Jugadores | 1 jugador |
| Compañía | Karoshi Corporation |
| Autores | Jon Cortazar, Eduardo Robsy, José Vicente Masó, Sergio Vaquer, Daniel Zorita, Javier Peña. |
| Distribuidora | Matra (a la venta por 14,95 €) |
| Trucos | POKE 34342, 201 - Energía infinita<br>POKE 33866, 201 - Tiempo infinito |

*Se busca héroe de piernas fornidas y excelente estado de forma para salvar país imaginario de maldición encantada. La recompensa, princesa forrada hasta el tuétano. Razón: Karoshi corporation. ¿Te suena el argumento? ¿Conoces a otros personajes cuya mayor habilidad es el salto largo y el salto corto? Si no es así, es que no has tenido un Spectrum, y me gustaría saber cómo narices has podido terminar leyendo este número de Magazine ZX.*

Así es, "La corona encantada" es un homenaje más a un subgénero prácticamente endémico de la Iberia profunda, hecho con amor, con mucho amor. Una historia de épocas pretéritas, con princesa encantada y objeto mágico para deshacer el hechizo que la mantiene presa. Ingredientes típicos de la estructura del cuento maravilloso, que Vladimir Propp ya diseccionara en su momento, y que tan bien se han adaptado al mundo del videojuego. Corre por bosques, grutas y castillos. Salta por extrañas islas flotantes, consigue las 20 monedas que dan acceso a la corona, y ve en busca de tu amada. El amor (y el dinero) te esperan si consigues hacerlo todo en el tiempo establecido.




Y ahora alguien me dirá: ¡pero es lo de siempre! Así es. Es lo que tienen los homenajes, que debes mantener las raíces para que sean reconocibles, y además, respetuosos. Aunque esto segundo quizás no se cumpla del todo. El buen hacer de Jon Cortázar ha dejado en evidencia los defectos que en su momento perdonamos a los iniciadores. Los Phantomas, Abu simbel y similares, palidecen ante las virtudes que en su momento no se

supieron (o no se pudieron) incluir en el código. Es la ventaja de crear a toro pasado, que ya sabes donde meter la mano (y donde no). La corona encantada cuenta con una coherencia absoluta en todo el apartado gráfico. Son poquitos píxeles, pero se nos muestra un mundo imaginario creíble, con continuidad. La sencillez de la imaginería que permiten 48K, no pervierte el elenco de razas y objetos enemigos que entorpecen nuestra labor, aún con su comportamiento básico de inteligencia primigenia. ¿A quién no le gustará ser de nuevo Bastian, a pesar de rondar o superar las tres décadas? Aquí es posible.

Siguiendo con los parabienes del juego, hay que destacar la dificultad ajustada del mismo. Un acierto que abre las puertas a todos los públicos y no sólo a entes con reflejos biónicos, y que desmitifica la endemoniada imposibilidad como mérito para crear juegos donde alcanzaremos a ver una sóla pantalla. Lo sabemos: nunca llegaremos al otro extremo, ni ganas que hay. El mapeado está estudiado al milímetro y racionado con un sistema de palancas que nos va abriendo nuevos caminos. Todo casa, todo funciona, sin cambios de escenario desorientadores. Todo invita a conseguir esa moneda escondida a un salto largo de distancia milimétrico. Si no es ahora, será en la siguiente partida. Tenemos energía de sobra para pensar en que el rescate de la princesa no es una utopía, y en todo caso, el vil dinero nos mantendrá con un hilo de vida. Si no lo perdemos intentando cogerlo, claro.

No obstante, hay que hacer notar algunas lagunas en el debe de la Corona encantada. La carga en modelos de 48K deja el juego completamente mudo, sin siquiera un mínimo zumbido surgido del z80. En ordenadores de más prestaciones cuenta con las voces del AY-3-8910 para todo el apartado sonoro. La música nos acompaña en la aventura, pero termina volviéndose monótona. Ya saben, por pedir que no quede, y se echa de menos un poquito más de variedad para que no terminen ahogándonos las limitaciones del chip de sonido y quemando una buena pero ermitaña orquestación. Otro pecado venial es la ausencia de teclas de pausa y vuelta al menú. Al menos, yo no las he encontrado. Lo de la pausa sí que precisa de penitencia, puesto que a pesar de que nuestro personaje no esté en peligro, el tiempo no deja de correr, y nos puede pasar factura al margen de la del teléfono del amigo que nos llama en medio de la partida.

La corona encantada es un desarrollo tanto para MSX como para Spectrum. Es muy posible que las limitaciones que he comentado sean una causa directa del hermanamiento, y de las fechas de entrega para llegar a tiempo a la convocatoria de la "MSXdev'08". Aún así, tiene el mérito de conseguir que volvamos a disfrutar de un género en el que ni la nostalgia nos impide pensar que ya lo hemos visto todo. Y eso, es algo muy difícil de conseguir. Además, supone el estreno de Karoshi corporation en las máquinas Sinclair, por lo que se impone dar la bienvenida a uno de los mejores desarrolladores del panorama actual de 8 bits. Esperemos que haya vendido cara su alma al diablo de Cambridge, y la deuda se vea saldada a base de más juegos.




**El regreso de Azpiri**

La Corona Encantada, sin quererlo, se ha convertido en el hasta ahora más ambicioso proyecto del desarrollo amateur de los ordenadores de 8 bits en España. Pensado para MSX y Spectrum, ha contado con la colaboración del prolífico Alfonso Azpiri para la carátula de la versión física. A Jon se le pasó por la cabeza, lo que a unos cuantos de nosotros: el tantear al ilustrador para tener el juego con una portada suya. Una empresa difícil de mantener con las ridículas cifras que se mueven hoy en este mundillo y que sólo es posible con un loco que financie semejante empresa. Afortunadamente, Karoshi cuenta con el soporte de la corporación MATRA, que se lió la manta a la cabeza y decidió embarcarse de nuevo en el barco pirata a la búsqueda del tesoro.

Azpiri ha disfrutado rememorando los viejos tiempos, y su colaboración no se ha visto exclusivamente reducida a la ilustración de portada, sino que también ha sido parte implicada del diseño: argumento, personajes, decorados,… La idea original también evolucionó gracias a él a lo que hoy tenemos entre manos los afortunados que cuentan con la cinta o el cartucho.

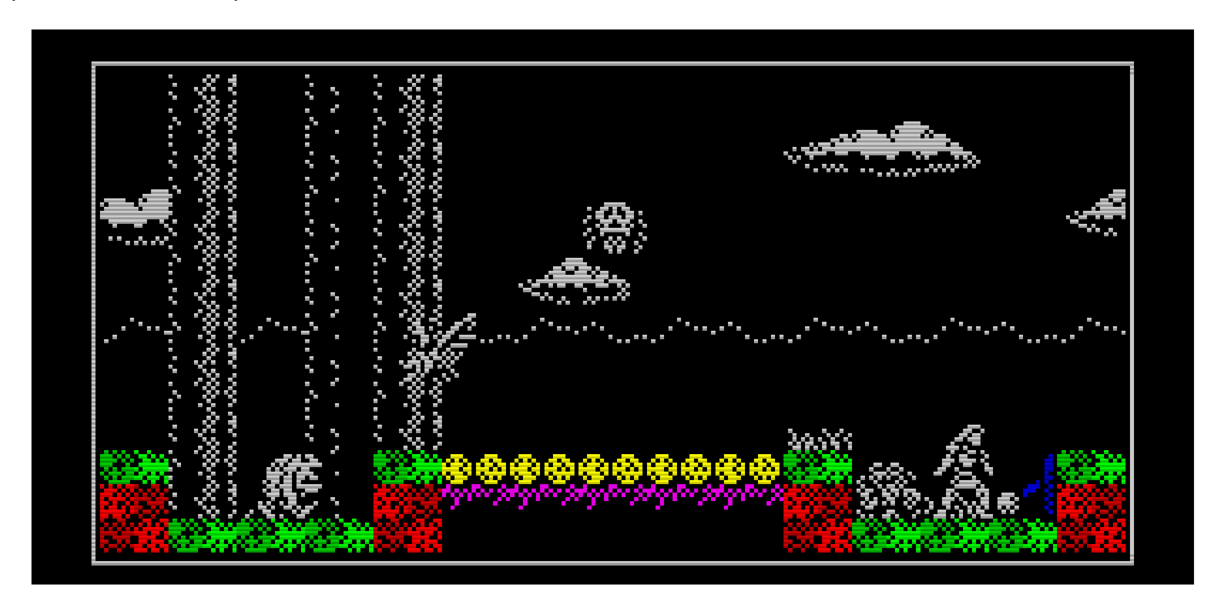



Valoraciones

Originalidad: [5]
Gráficos: [8]
Sonido: [7]
Jugabilidad: [8]
Adicción: [8]
Dificultad: [7]

**La Corona, al descubierto**

Comienza nuestro viaje en busca de las veinte monedas. Aparte de los múltiples enemigos, encontraremos palancas que nos permitirán acceder a diferentes partes del escenario. Encaminaremos nuestros pasos hacia la derecha para accionar la primera de ellas.





Volvemos a la izquierda, hacia la nueva plataforma que ha aparecido, con cuidado al saltar por encima de un agujero que nos provocará la muerte segura si caemos en él. Nos encaramamos a la plataforma y volvemos a la derecha, pero siempre ascendiendo. Una pantalla a la derecha y otra hacia arriba volvemos a la izquierda (subiendo), para accionar una nueva palanca. En el camino vemos una moneda, pero enseguida volveremos hacia ella.

Esta vez la palanca nos franqueará el acceso a los niveles inferiores, en los que se encuentra retenida la princesa. De vuelta hacia la derecha, nos dejaremos caer de la plataforma, para recoger la primera moneda de nuestro botín, que se encuentra justo debajo, y volvemos a retomar la ruta ascendente hacia la derecha, para recoger la segunda moneda (la que vimos antes).

Aquí podemos encaramarnos a la plataforma central si dejamos que el pájaro vaya hacia la izquierda y hacemos un salto alto justo en ese momento.



En la siguiente pantalla de la derecha, nos dejaremos caer justo por el centro, para recoger la moneda (3) que se encuentra, una vez más, justo debajo. Nuevamente, repetiremos el proceso de ascensión, por tercera vez, hasta retornar a la pantalla anterior.



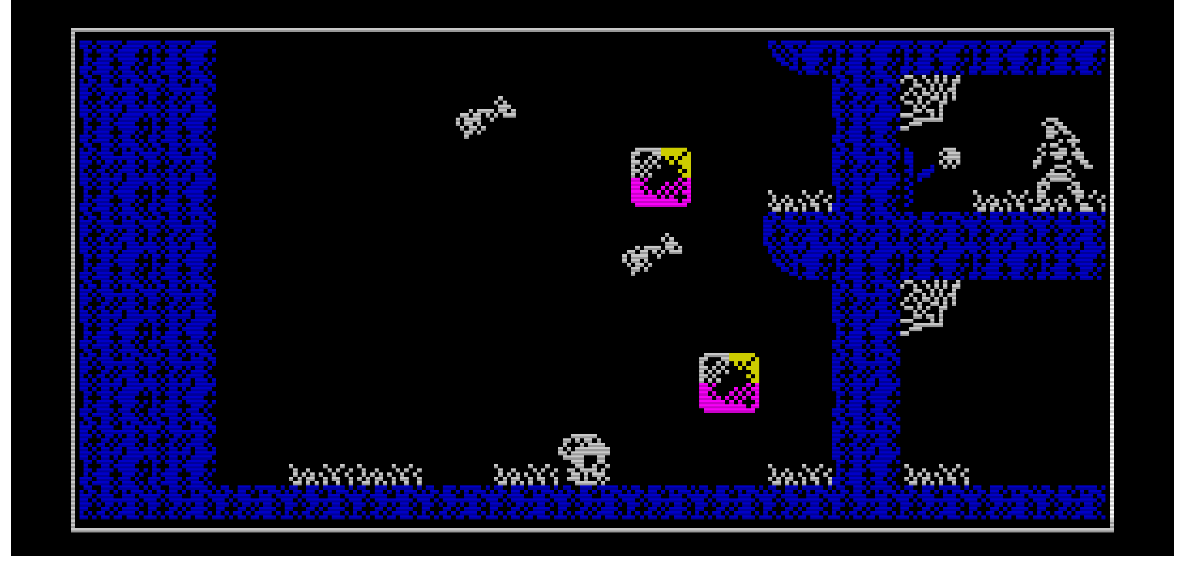
Continuamos hacia la derecha. En la siguiente pantalla, hay una moneda (4) en el lado derecho. Para acceder a ella, además de esquivar a los enemigos, deberemos realizar con precisión un salto largo. Nos dejamos caer a plomo de esa plataforma para encontrar otra justo debajo, con una nueva moneda. Vamos hacia la derecha y, justo al borde de la plataforma, ejecutamos un salto largo que nos hará caer en otra desde la cual podremos accionar una palanca, que hará aparecer un par de piedras que nos darán acceso, más adelante, a una moneda (5) en la fachada del castillo.





Lo siguiente que haremos será dejarnos caer y adentrarnos en las profundidades, ya que todavía no nos es posible acceder al castillo donde se encuentra la corona. Nada más entrar en la cueva encontraremos una nueva moneda (6). Nos encaminamos a la izquierda y, en la siguiente pantalla, nos dejamos caer para accionar una palanca que nos permitirá el paso por la parte superior.

Volvemos a subir, seguimos hacia la izquierda, recogemos la moneda (7) y bajamos por el hueco que acabamos de habilitar. Debajo encontraremos otra moneda (8) y podremos seguir hacia la izquierda. En la siguiente pantalla, nos dejaremos caer y continuaremos en la misma dirección, yendo por la parte de arriba de la pantalla, para activar una palanca. Ésta es la que nos permitirá acceder al castillo.



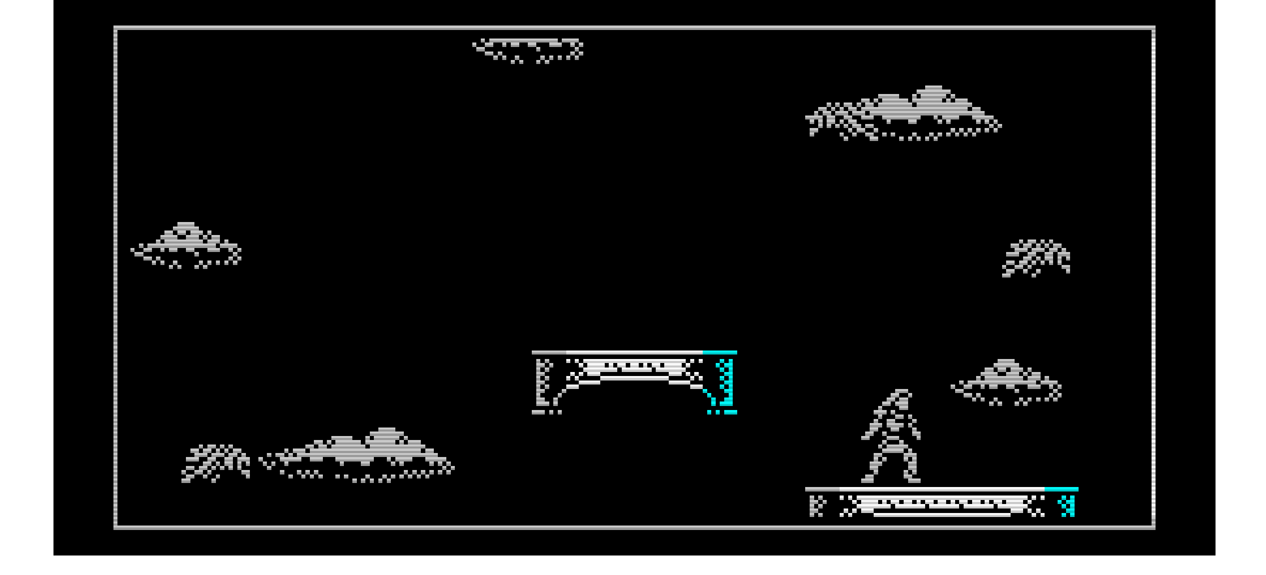


Pero todavía no podemos ir hacia allá, ya que tenemos que recoger todas las monedas y todavía nos quedan algunas. Volvamos a la pantalla superior y, esta vez, continuaremos hacia arriba. Recogemos la moneda (9) y viramos a la izquierda. Al final de ese camino está la princesa, pero sin la corona, no podremos liberarla. Nos dejaremos caer por la zona izquierda y proseguiremos por ahí. Cuando lleguemos al final, nos dejamos caer y llegaremos al final de la cueva por ese lado, donde nos espera una moneda (10). De vuelta, recogeremos otra moneda (11) que hemos dejado atrás justo antes. Hecho lo cual, podemos encaminar nuestros pasos a la superficie, hacia el castillo, que se encuentra a la derecha. Para cruzar el acceso a las cuevas deberemos ejecutar con precisión un salto largo.

Accedemos al castillo esquivando al ogro feo que custodia la entrada y nos dirigiremos primero hacia las estancias superiores. Recogemos la moneda (12) ubicada en el borde derecho y continuamos hacia la izquierda. En la siguiente pantalla, podremos ascender hacia las murallas. Recolectaremos primeramente la moneda (13) situada en la parte derecha.

Volvemos nuestros pasos hacia la izquierda y nos dejaremos caer justo al borde de la muralla, para ir a caer a las piedras que mostró una de las palancas y poder recoger la moneda (14). Desde aquí, no nos queda otra que dejarnos caer (menos mal que nuestro protagonista parece un gato y no le afectan las caídas verticales desde mucha altura) y volver al castillo. Repetimos nuestros pasos y volvemos a las murallas.

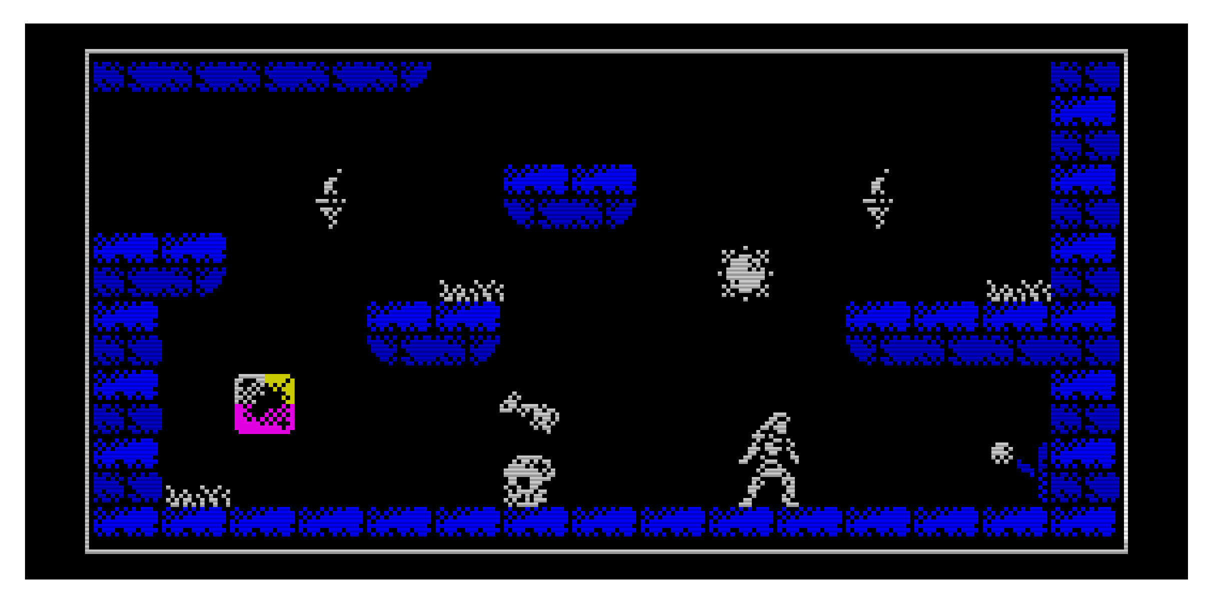
En el borde izquierdo del castillo veremos una nueva moneda (15) en una plataforma ubicada en el cielo, a la que accederemos desde las almenas. Seguimos hacia la izquierda y, con un salto largo (y de fe) accederemos a otras plataformas, en las que encontraremos la última moneda (16) de esta zona.

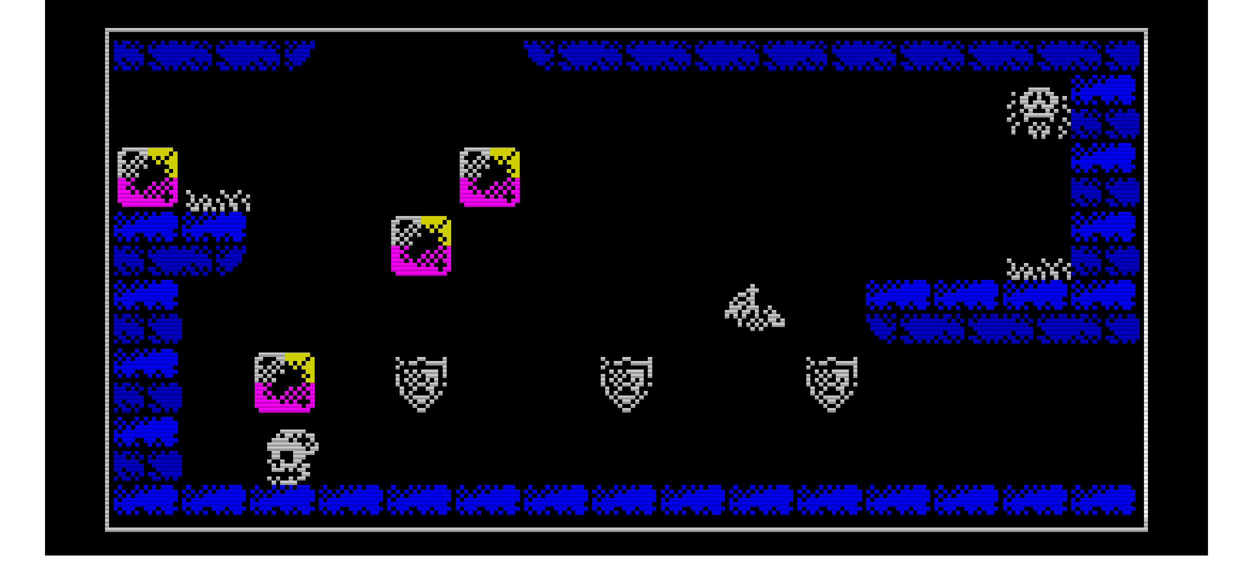


Por tercera y última vez entraremos al castillo y nos dirigiremos hacia la zona inferior, en busca del Consejero Real. Nada más bajar encontraremos una nueva moneda (17). Nos encaminamos hacia la izquierda y la siguiente pantalla la recorremos por la zona superior, para dejarnos caer pegados al borde izquierdo, recogiendo así una nueva moneda (18).

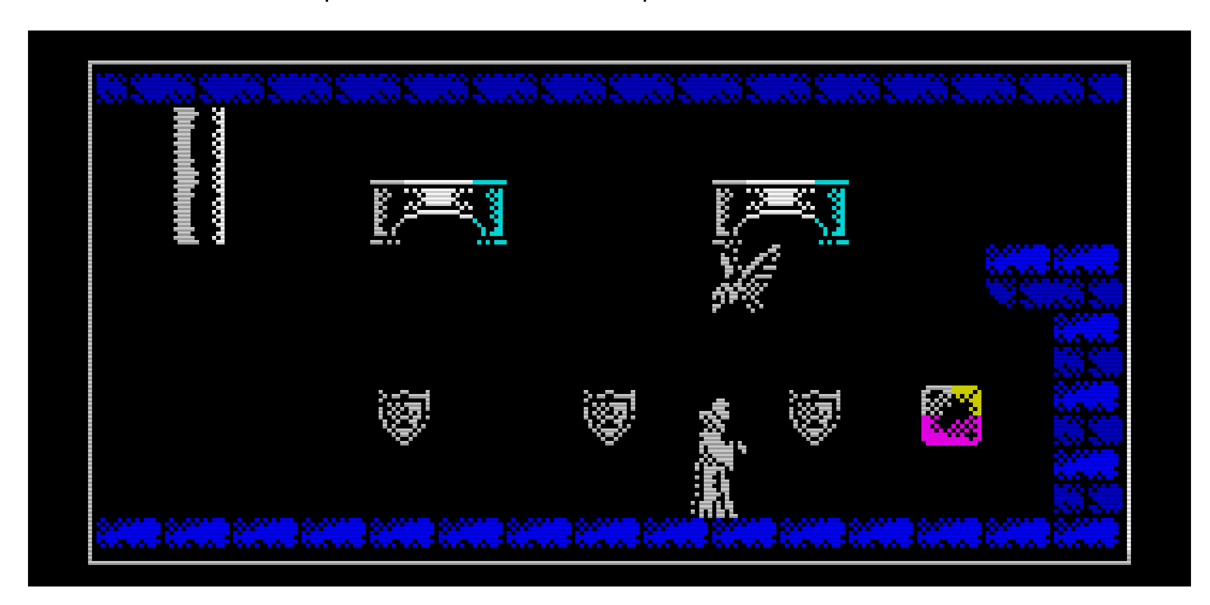


Nos dejamos caer y seguimos hacia la izquierda y abajo hasta que no podamos continuar. En ese punto hay unas piedras que bloquean el acceso a la sala de la corona, así que busquemos la palanca que nos dará acceso, que está hacia la derecha. De camino, recogeremos otra moneda (19). Antes (o después) de accionar la palanca, podremos recoger la última moneda (20) de la sala inmediatamente superior.





¡Ya tenemos las 20 monedas! Así que tendremos vía libre para acceder a la corona.



Volvemos sobre nuestros pasos hacia la izquierda. Pasamos por la estancia en la que se encuentra el Consejero Real (del que daríamos cuenta de buena gana, pero en este juego no hay ni espadas ni posibilidad de agacharse, recordad), así que nos hacemos directamente con la corona. ¡La princesa no puede esperar!





No obstante, salvo que vayamos muy justos de tiempo, ahora lo importante es no perder la paciencia, ya que lo más difícil está hecho, y todas las pantallas por las que pasaremos ya las hemos superado anteriormente. Si somos capaces de dominar los nervios, salir del castillo, volver a la cueva y llegar hasta la princesa, podremos contemplar el ansiado final.

---

Descarga el mapa de la aventura: mapa_corona.png

---

Javier Vispe y Federico J. Álvarez

# ZX BASIC. Un sueño hecho real...

*Este artículo trata sobre ZX BASIC, un compilador cruzado que permite crear en tu PC programas para tu ZX Spectrum. Lo que hace es traducir las instrucciones de un programa en BASIC a código máquina del Z80 que luego puedes ejecutar en un emulador o en un Spectrum real.*

**Hace mucho tiempo...**

Ejem... ¿En una galaxia muy muy lejana? Bueno, no. No tanto.

Al igual que muchos de vosotros, de niño quería hacer otras cosas con mi ZX Spectrum aparte de jugar: quería experimentar. Programar era una forma de hacerlo, sin duda. La electrónica no era mi fuerte y siendo un crío como era y con tan poca experiencia (había tenido algún que otro susto con la electricidad anteriormente) la programación era la mejor forma de experimentar. Era como ser un dios en miniatura dentro del universo de posibilidades que el ZX Spectrum ofrecía por aquel entonces. Compraba MicroHobby (y las primeras Micromanía); me dedicaba -como creo que todos hicimos- horas y horas a teclear los listados publicados; a hacer mis propios códigos en BASIC; a cargar algunos de los programas y modificarlos (hoy diríamos hackearlos) y en definitiva a probar muchas otras cosas de esta pequeña pero maravillosa máquina.

Pronto una limitación se hizo evidente: el BASIC del ZX Spectrum era ciertamente lento. Cuando nos picábamos mis amigos y yo con nuestros respectivmos micros, el ZX Spectrum siempre llevaba las de perder. Un simple bucle FOR n = 1 TO 1000: NEXT n tardaba algo más de 4 segundos en ejecutarse aunque no hiciera nada. En otros microordenadores esto no era así (en el Dragon 64 tardaba menos de un segundo, creo recordar).

Pese a todo, el BASIC de Sinclair era extremadamente rico en general. Aprendí programación con él y muchas de esas cosas me ayudaron en la carrera de informática. Esa riqueza del lenguaje no parecían tenerla los otras implementaciones BASIC de los años 80 (o esa fue mi impresión). Un ejemplo típico es el COMMODORE, cuyo BASIC emplea muchos POKES para realizar determinadas tareas. Es por esto que creo que es uno de los micros con mayor cantidad de programas BASIC creados (todavía a día de hoy).

Para vencer la limitación de la velocidad, muchos intentamos programar en ensamblador. Mis escasos conocimientos en general y particularmente sobre todo de aritmética binaria y hexadecimal, la escasa bibliografía -gracias de nuevo a MicroHobby y a sus fichas- y, sobre todo, de un ensamblador decente (nunca tuve GENS ni MONS) hicieron que el sueño de crear un juego y otras muchas cosas nunca se cumplieran.

Llegué a hacer en BASIC un intérprete de lenguaje Prolog muy simple. Otro de LOGO y otro de un lenguaje que me inventé. Desde luego, no tenía el conocimiento sobre la programación de lenguajes de ordenador que tengo hoy día, pero como ya os imagináis, si el BASIC de ZX era lento, un intérprete de un lenguaje realizado en BASIC era aún más lento. Exasperante. A pesar de todo, esa pequeña frustración fue la semilla que fijó claramente mi vocación (como supongo que muchos de vosotros).

Si estás leyendo esto es porque muy probablemente también tuviste estas inquietudes. Ahora volvamos al presente...

**Un poco de teoría**

Antes de comenzar a explicar el uso del compilador, está bien echar un vistazo por encima a qué es un compilador y su diseño interno por dos motivos:

- Hay personas que saben programar pero no comprenden que hay cosas que un compilador no podrá hacer jamás.
- También puede que te interese expandir el compilador de alguna manera, o contribuir al desarrollo del lenguaje o a muchas otras cosas.

Para empezar, un traductor no es más que un programa capaz de convertir de alguna manera un código escrito en un lenguaje de programación a otro. Por ejemplo puedes tener un programa en BASIC y traducirlo a LOGO (suponiendo que sea factible) o a C. Un compilador va un paso más allá. Es un traductor que traduce un lenguaje fuente a código objeto (binario) que es directamente ejecutable por la máquina.

Un intérprete, por contra, lo que hace es ir leyendo un programa fuente instrucción por instrucción y representarlas ejecutando una serie de acciones equivalentes a cada una de ellas. Si usaste un ZX Spectrum ya conoces un intérprete: la ROM del ZX Spectrum. La ROM es en su mayor parte un intérprete de Sinclair

BASIC. Los programas interpretados, por lo general, son más lentos que los compilados (en algunos pocos casos se puede conseguir una velocidad casi similar).

ZX BASIC es un compilador cruzado. Los compiladores cruzados son aquellos que se ejecutan en una máquina pero producen código objeto para otra. En este caso, vamos a usar ZX BASIC en nuestro PC (ya sea sobre Windows o Linux) para producir un programa compilado para ZX Spectrum (en realidad para Z80). De esa manera ahorraremos memoria -que en el Spectrum es muy escasa- al no tener que alojar el compilador junto con el programa compilado. Además, la velocidad de compilación será mayor (generalmente unos segundos).

La velocidad de un programa compilado estriba en que se calcula de antemano toda la información posible sobre el código fuente que se va a ejecutar. Por eso un compilador necesita obtener determinada información sobre el programa durante la fase de compilación (por ejemplo, el tipo de dato que almacena una variable, si es numérica o de cadena, dónde se va a guardar en memoria, etc). Un intérprete no necesita esto ya que lo hace durante la ejecución. Por eso hay cosas que un compilador no puede hacer y un intérprete sí. El ejemplo más claro es el de la función VAL del Sinclair BASIC. Es una instrucción muy potente. LET a$ = "x+x": LET b = VAL a$ almacena en la variable b el resultado de la expresión "x+x". Dado que la variable a$ puede cambiar su valor durante la ejecución del programa, es imposible saber en tiempo de compilación que tipo de dato se va a guardar en b (un valor de punto flotante, un número entero, etc) ni cómo calcularlo. De hecho, muy pocas implementaciones de BASIC tienen esta potencia, ni siquiera en la actualidad. En las competiciones de BASIC entre distintas marcas de micros antes mencionadas, el ZX perdía en velocidad pero por contra la capacidad de VAL resultaba devastadora.

En definitiva, esto era posible porque el BASIC de la ROM cada vez que evalúa una expresión le hace un análisis (con la consiguiente lentitud): se gana versatilidad, pero se pierde velocidad.

Los compiladores modernos (y ZX BASIC lo es) se basan en capas:

- La primera capa es la de análisis de código (que a su vez se suele descomponer en dos: una capa de análisis léxico y otra de análisis sintáctico). Esta capa intenta comprender el programa y verificar que la sintaxis es correcta construyendo una representación en memoria del programa llamada Árbol Sintáctico Abstracto (AST en inglés).
- La segunda capa es la de análisis semántico, que realiza una primera traducción y además algunas comprobaciones extra que la capa anterior no puede realizar. También se realizan las primeras optimaciones de código. En general, lo que hace es traducir el programa a un ensamblador ficticio llamado código intermedio. Esta capa y la anterior suelen ir juntas en el código (muchas veces son la misma capa en realidad) y se les llama frontend.
- Este código intermedio será nuevamente traducido a ensamblador de la arquitectura de destino (en nuestro caso Z80). Al contrario que el lenguaje BASIC, el código intermedio es muy rígido y muy fácil de analizar. No es necesario crear otro traductor para esta fase. A esta capa y a las que siguen se las suele llamar backend.
- Finalmente, un ensamblador (el ZX BASIC contiene uno propio) y un enlazador de código objeto (el ZX BASIC no usa ninguno: por ahora todo se trabaja en ensamblador y se compila a binario directamente) realizan el resto del trabajo produciendo el binario final.

| | | |
|---|---|---|
| Frontend | Análisis Léxico | Convertir letras a palabras y símbolos |
| | Análisis Sintáctico | Comprobar la sintaxis y las frases |
| | Análisis Semántico | Comprobar los tipos de variable, declaraciones duplicadas o fuera de contexto. Construcción del Árbol Sintáctico. |
| | Optimización del Árbol | |
| | Generación de Código Intermedio | |
| Backend para ZX Spectrum | Traducción a Ensamblador (Z80) | |
| | Optimización de Código Ensamblador (reordenación de registros, etc) | |
| | Ensamblado: Traducción a Código Máquina (o Código Objeto) | |

Prácticamente todos los compiladores actuales trabajan de forma similar. La ventaja de esto es que se pueden cambiar las capas de backend de manera que es posible compilar el mismo programa para distintas arquitecturas: se podría hacer que un programa en ZX BASIC compilara para Windows, Linux, Nintendo DS, teléfono móvil, PlayStation o cualquier otra plataforma. Evidentemente cada plataforma tiene su limitación, pero dado que el ZX es la más limitada de todas sin duda, esto no debería suponer problema alguno. Los compiladores de este tipo se llaman retargeable compilers (compiladores reorientables).

También se podría cambiar el frontend. Podrías crear un traductor para otro lenguaje (C, PASCAL, LOGO...) que pasara a código intermedio, y usar el mismo backend. Esta filosofía es la que usa .NET, por ejemplo, donde distintos lenguajes fuentes compilan a CIL (Common Intermediate Language) que es luego interpretado por el .NET Framework que tengas instalado en Windows (si es que usas ese sistema operativo).

Resumiendo: ZX BASIC es un compilador cruzado reorientable de 3 capas.

**Instalación**

El compilador ZX BASIC está hecho en Python (un lenguaje interpretado de scripting). Esto lo hace portable a todas aquellas plataformas donde esté python (actualmente incluso para telefonos Nokia con Symbian, y para Windows Mobile). Es posible en teoría compilar programas en ZX BASIC para nuestro Spectrum en cualquier sitio donde esté python. Lo único que se requiere es que la versión de Python instalada sea igual o superior a la 2.5. En el caso de Windows, además, hay una versión instalable (.MSI) que ni siquiera requiere python. Si usas Windows y tienes dudas, te recomiendo que uses esta versión.

Puedes descargarte la última versión del compilador desde http://www.boriel.com/files/zxb/ . Verás que hay varias versiones. Descárgate siempre la más reciente. Hay un alias llamado latest version que siempre apunta a la última versión (en el momento de escribir esto, la 1.0.3). El archivo con extensión .MSI es el archivo de instalación de Windows antes mencionado. Si no estás familiarizado con todo lo que has leído hasta ahora, usa esta versión.

Para Linux y otras plataformas, descomprime la versión .zip (o .tar.gz, la que te sea más cómoda) en algún sitio de tu PATH. Para desinstalar el compilador, sólo tienes que borrar la carpeta donde lo has descomprimido. Si usaste la versión .MSI de windows, desinstálalo desde el Panel de Control (Agregar o Quitar Programas).

Para comprobar que se ha instalado correctamente, abre una consola y teclea zxb.py --version. Si usaste la versión autoinstalable de Windows, abre una ventana nueva de consola de comandos MS-DOS porque las variables de entorno han cambiado, y teclea: zxb --version (fíjate que no tiene la extensión .py). En ambos casos, se debería imprimir la versión del compilador (1.0.5).

A partir de ahora todos los ejemplos usarán la orden zxb (usa zxb.py si instalaste el compilador descomprimiéndolo en una carpeta).

**Nuestro primer programa**

El ritual a la hora de probar un nuevo lenguaje es el programa Hola Mundo, que básicamente imprime ese mensaje por pantalla. Vamos a hacer lo mismo, para familiarizarnos con todo el proceso de creación y compilación de un programa.

El código fuente de ZX BASIC tiene que estar en un archivo de texto ASCII. Usa tu editor de texto favorito, y escribe el siguiente programa:

```
10 PRINT "Hola Mundo!"
```

Como puedes ver, es una sentencia BASIC de toda la vida. Ahora graba el programa como hola.bas. Nunca grabes tus programas en el directorio de instalación del compilador. Hazlo en un directorio propio y evita que tu código se mezcle con el del compilador. De esa forma, cuando salgan nuevas versiones no tendrás problemas borrando el directorio del compilador para instalar la nueva versión.

Ahora abre una consola de comandos y posiciónate en el directorio donde has grabado el programa hola.bas. Vamos a realizar la compilación:

```
zxb -T -B -a hola.bas
```

Te recomiendo que dejes de leer aquí y hagas todo esto en el computador antes de seguir. Si todo ha ido bien no aparecerá nada por pantalla. Se tiene que haber creado el programa hola.tzx (que es tu programa compilado en formato .tzx) en el mismo directorio donde tienes el código fuente hola.bas. Este programa se puede ejecutar directamente en cualquier emulador (yo utilizo EMUZWIN) que admite el formato .TZX. También hay programas que convierten los archivos .TZX a archivos de sonido .MP3 que son los que se usarían para cargar en un ZX Spectrum real. De todas formas se pueden generar otros formatos.

Recuerda que tu programa es compilado en código máquina. Normalmente el ZX Spectrum necesita un cargador en BASIC que cargue el código máquina y lo ejecute. Si cargas el programa a velocidad normal verás que primero aparecerá en pantalla Program: "hola.bas". Es el cargador en BASIC. Este cargador a continuación cargará el código máquina y lo ejecutará con un RANDOMIZE USR. Si interrumpes la carga verás que es un cargador muy simple y sin ninguna protección y que el código máquina de tu programa se carga y ejecuta a partir de la dirección de memoria 32768 (8000h en hexadecimal).

Si todo esto te ha funcionado, enhorabuena: ya estás listo para empezar a programar. ZX BASIC intenta ser lo más parecido al Sinclair BASIC, de manera que te sea cómodo empezar a programar si ya conoces éste último. Sin embargo, existen algunas limitaciones por ser un programa compilado (como ya se comentó antes). Algunas instrucciones como VAL no se comportan igual. Otras no existen porque no tienen sentido (LIST, LLIST). Y algunas otras están expandidas o admiten parámetros extra para aprovechar la potencia que ahora tenemos. Además, hay instrucciones nuevas que te ahorrarán trabajo: sentencias de bucles más potentes y subrutinas con variables privadas son sólo un ejemplo de ello.

**Parámetros de compilación**

El compilador ZX BASIC, por defecto, lee el código fuente y genera un archivo binario con la extensión .bin que

contiene el código máquina (es decir, los bytes del código objeto directamente). Este código máquina no es reubicable, y tiene que ser cargado a partir de la dirección 32768. Como puedes suponer, esto puede ser de poca utilidad si no disponemos de algún método para pasar este código máquina a un emulador o un ZX Spectrum real. Además, puede que queramos que nuestro código empiece en la dirección 28000 (para ganar algo más de memoria que tan escasa es en nuestro ZX). Todo esto y mucho más se puede cambiar con las opciones de compilación.

ZX Basic tiene bastantes parámetros. Sólo veremos los más importantes. Si tecleas zxb -h tendrás una escueta ayuda. Puedes buscar más información en la Wiki del compilador, que está en http://www.boriel.com/wiki/en/index.php/ZXBasic en inglés. Los parámetros tienen una versión larga que empieza con doble guión (por ej. --tzx) y una versión corta equivalente (en este caso, -T). Los parámetros más importantes son:

- -T o --tzx hace que el formato de salida sea un archivo de cinta .tzx
- -t o --tap hace que el formato de salida sea un archivo de cinta .tap. Esta opción y la anterior son excluyentes.
- -B o --BASIC hace que se genere un cargador BASIC que cargue nuestro programa. Si usas esta opción, necesariamente tienes que haber usado una de las anteriores.
- -a o --autorun hace que nuestro programa se ejecute automáticamente tras ser cargado. Esta opción obliga a que se use la opción --BASIC, ya que se requiere un cargador BASIC para que el programa se autoejecute.
- -o <fichero de salida>. Permite especificar un nombre de archivo distinto para el programa resultante. Generalmente se usará el mismo nombre que el programa de código fuente (.bas), pero con una extensión distinta (.bin, .tap o .tzx)
- -A o --asm Hace que no se genere el código objeto. En lugar de eso obtendremos el código ensamblador de todo nuestro programa (un archivo ASCII con extensión .asm). Esta opción es útil si queremos optimizar a mano nuestro programa. Se ha intentado que el código ensamblador generado sea bastante legible y con comentarios incluidos.
- -S o --ORG cambia la dirección de comienzo de ejecución del código máquina. Como se dijo antes, por defecto es 32768, pero podemos hacer que se ejecute a partir de una dirección distinta (por ej. 28000) para obtener más memoria.

**El Lenguaje ZX BASIC**

He intentado que el lenguaje ZX BASIC sea lo más similar posible al de Sinclair. Como ya expliqué en el apartado anterior, hay instrucciones que no tienen sentido en un programa compilado y otras que son prácticamente imposibles de implementar. Pero también es cierto que se puede expandir el lenguaje y crear nuevas instrucciones para hacer cosas con la potencia que ahora tenemos. Para llegar a un consenso, se han tomado palabras reservadas del lenguaje FreeBASIC. En general, lo mejor es ir a la Wiki del compilador ZX BASIC ya mencionada pues allí hay una referencia de las palabras clave así como multitud de ejemplos.

Algunas cosas importantes:

- ZX BASIC distingue en entre mayúsculas y minúsculas en los nombres de variables y funciones. Una variable llamada A es diferente de otra llamada a. Sin embargo en lo que se refiere a palabras reservadas (por ejemplo PRINT o FOR), éstas se pueden escribir como se deseen. La distinción entre mayúsculas y minúsculas para los identificadores de usuario podrá ser configurable en futuras versiones del compilador, pero no actualmente.

  Ejemplo.- Las siguientes palabras reservadas son todas equivalentes

  ```
  PRINT print Print PrInT
  ```

- Como en el Spectrum, el ZX BASIC está orientado a líneas: no puedes partir una línea en medio de una sentencia BASIC. Si deseas hacerlo, usa el carácter de subrayado (_), al final de la línea partida para indicar que ésta continúa en la siguiente.

  Ejemplo.- La siguiente línea partida no dará error, porque lleva un caracter de subguión (o subrayado) al final

  ```
  10 PRINT _
      "Hola Mundo"
  ```

- Los números de línea del BASIC ya no son obligatorios. Son opcionales y se usan como etiquetas. También puedes usar identificadores como etiquetas. De todas maneras, si quieres puedes seguir usando líneas numeradas como en el BASIC tradicional del ZX Spectrum, si te puede la nostalgia (a mí a veces me puede, qué demonios). ¡Además, ya no están limitadas a 9999, y ni si quiera tienen por qué estar numeradas en orden!

  Ejemplo.- Un bucle infinito sin usar numeración de líneas:

```
BucleInfinito: REM Esto es una etiqueta.
       PRINT "Hola Mundo" : REM Esto es una línea sin numeración
       GO TO BucleInfinito : REM Esto podría ser GO TO 10 en Sinclair BASIC
```

**Los tipos de datos**

Si eres un programador de Sinclair BASIC con cierta experiencia seguramente sabrás que el BASIC del ZX Spectrum trabaja siempre en punto flotante. El formato de dicha representación puede resultar complejo y ocupa 5 preciosos bytes (una mantisa de 32 bits y un exponente de 8 en exceso 127, tal y como explica el manual del ZX). Eso se traduce en un coste enorme en tiempo y en memoria. Si solo queremos almacenar valores enteros menores que 255, podríamos usar bytes directamente. Ocuparían la quinta parte de la memoria y su procesado sería mucho más rápido (apenas una o dos instrucciones en ensamblador). Por contra, se puede perder precisión en algunos cálculos. No te preocupes: el formato de punto flotante también está disponible.

Está claro que según el uso que le vayamos a dar a una variable, ésta almacenará un tipo de dato determinado. Este tipo de dato le indica al compilador cómo tratarlo y el tamaño que ocupará en memoria. Así pues, usaremos tipos de datos y diremos que ZX BASIC es un lenguaje tipado. Los tipos de datos que maneja el ZX BASIC son:

| **Tipo** | **Tamaño** | **Clase de dato** | **Intervalo** |
|---|---|---|---|
| Byte | 1 byte | Entero corto con signo | [-128 ... 127] |
| UByte | 1 byte | Entero corto sin signo | [0 ... 255] |
| Integer | 2 bytes | Entero con signo | [-32768 ... 32767] |
| Uinteger | 2 bytes | Entero sin signo | [0 ... 65536] |
| Long | 4 bytes | Entero largo con signo | [-2147483648 a 2147483647] (más de 2.100 millones) |
| Ulong | 4 bytes | Entero largo sin signo | [0 .. 4294967295] (de 0 a más de 4.200 millones) |
| Fixed | 4 bytes | Decimal con punto fijo | [-32767.9999847 .. 32767.9999847] con una precisión de 1 / 2^16 (0.000015 aprox.) |
| Float | 5 bytes | Decimal con punto flotante | Como en el ZX Spectrum |

Estos son todos los tipos de datos numéricos. Los enteros deberían estar claros. Además hay un tipo de decimal en punto fijo, llamado Fixed, muy interesante. Si vas a usar números decimales entre -32767 y 32767 quizá deberías usar este tipo de números (siempre y cuando sólo uses las 4 operaciones básicas: sumas, restas, productos y divisiones).

Estos tipos de datos se usan para almacenar valores y computar expresiones matemáticas. Las expresiones matemáticas usan la misma sintaxis que el Sinclair BASIC y los mismos operadores y precedencia, con la excepción del operador de potencia (^). La expresión 2^2^3 se calcula de forma distinta en un Spectrum que en la mayoría de los lenguajes. Haz la prueba... En cualqueir caso, si tienes dudas, usa paréntesis.

**Variables de Cadenas de Carácteres**

Al igual que Sinclair BASIC, ZX BASIC también es capaz de manejar cadenas alfanuméricas. Tradicionalmente, en el lenguaje BASIC las variables alfanuméricas se denotaban con el sufijo $. En ZX BASIC (y los BASIC modernos en general) esto es opcional. Además, Sinclair BASIC sólo permitía nombres de variables alfanuméricas de una letra. Aquí no existe esa limitación, pudiendo usar variables alfanuméricas como una_variable_con_nombre_muy_largo$.

Los valores alfanuméricos se pueden unir (concatenar) usando el operador de suma (+), al igual que en Basic de Sinclair.

**Asignando Valores a Variables**

**Al igual que en Sinclair BASIC, se usa la sentencia LET para asignar un valor a una variable. Pero ahora ésta se puede omitir. Las siguientes líneas son equivalentes:**

```
LET a = 1
a = 1
```

**DIM: Declarando Variables**

**Hemos visto que ZX BASIC usa distintos tipos de dato. Cuando usas una variable el compilador intenta adivinar su tipo (si es alfanumérica, entera, etc.), pero en general no va a poder hacerlo. Si no sabe qué tipo asignar, usará el punto flotante como el Sinclair Basic. Esto conlleva un desperdicio de memoria y mayor lentitud. Podemos indicarle al compilador el tipo de dato de una variable declarándola. Para declarar una variable se usa la palabra reservada DIM:**

```
REM Declaración de dos variables de tipo entero byte sin signo
DIM a, b uByte

REM Declaración de una variable en punto Flotante con valor inicializado
DIM longitud = 3.5 AS Float: REM longitud = 3.5 metros
```

**Si vas a declarar una variable, tienes que hacerlo antes de su primer uso.**

**Arrays**

**Los arrays se declaran como en Sinclair BASIC, y admiten tantas dimensiones como quepan en memoria. No obstante, al contrario que en Sinclair Basic, la numeración de los índices no comienza en 1, sino en 0. Además, podemos declarar opcionalmente el tipo de elemento. Veamos un ejemplo:**

```
DIM a(10) AS Float : REM un array de 11 flotantes, del 0 al 10, ambos inclusive
```

**Podemos trabajar normalmente como en Sinclair BASIC empezando desde índice uno, pero desperdiciaremos la posición 0. No obstante, también podemos declarar el índice mínimo y máximo del array de forma explícita:**

```
DIM a(1 TO 10, 1 TO 5) AS Float : REM  esto es equivalente a DIM a(10, 5) en el BASIC de ZX Spectrum
```

**Si queremos que por defecto los arrays comiencen en 1 (como en Sinclair BASIC) y no en 0, hay que usar la opción --array-base=1 al invocar al compilador.**

**Al contrario que en el BASIC de Sinclair, el nombre de las variables de array puede tener cualquier longitud (estaban limitados a una sola letra en el Spectrum, al igual que las variables de bucles FOR y las alfanuméricas). Existen muchas más cosas que se pueden hacer (como declarar arrays inicializados) pero no las veremos en este artículo. Pregunta en el foro de speccy.org o en del compilador, o mejor consulta la Wiki.**

**Es posible declarar arrays con valores inicializados. Esto es útil porque no existen ni READ, ni DATA, ni RESTORE. Lo haríamos así:**

```
REM Definimos un array de 2 filas y 8 columnas
DIM MiUdg(1, 7) AS uByte => { {60, 66, 129, 129, 129, 129, 66, 60}, _
                              {24, 60, 60, 60, 126, 251, 247, 126}}
```

**Observa el caracter de subrayado "_" al final de la primera línea del array, ya que hay que partirla (si no, quedaría muy larga).**

**Sentencias de Control de Flujo**

***GO TO, GO SUB, RETURN***

**Idénticas al BASIC de Sinclair, aunque se desaconseja su uso. GO TO puede escribirse junto, como GOTO. Ídem para GOSUB. Se pueden usar números de línea o etiquetas como vimos en un ejemplo anterior.**

***FOR***

**La sentencia para realizar bucles FOR se comporta de la misma forma que en ZX Basic. No obstante, al ser un lenguaje compilado, tienes que tener en cuenta algunas cosas. Por ejemplo, hacer un bucle FOR usando una variable en punto flotante es bastante más lento que hacerla con una de tipo entero. Su uso es prácticamente igual al del ZX Spectrum, pero hay algunas diferencias:**

- **Los nombres de las variables de bucle FOR pueden tener más de una letra (no así en Sinclair BASIC)**
- **El nombre de la variable se puede omitir en las sentencias NEXT. O sea, se puede escribir FOR x = 1 TO 10: NEXT : REM se omite la 'x' en NEXT**
- **No se puede poner un NEXT en cualquier parte del programa. Tiene que ser después de un FOR. Para bucles anidados (uno dentro de otro) el NEXT interno tiene que referirse obligatoriamente al bucle más interno. Ídem para los restantes bucles.**

**Existen además, dos sentencias de control de bucles nuevas:**

- **EXIT FOR termina el bucle FOR y salta justo al final (solíamos usar on GOTO para esto, ¿verdad?)**
- **CONTINUE FOR que "continúa" el bucle, es decir, realiza un NEXT. En el BASIC de Sinclair poníamos un NEXT <variable>, pero en ZX BASIC esto no se permite. Hay que usar CONTINUE.**

***IF***

**Esta sentencia sí difiere del BASIC tradicional de Sinclair. Es más potente. Admite varias líneas después del THEN, y además incluye la cláusula ELSE ("en otro caso"). Además es necesario terminarla con un END IF. Un ejemplo:**

```
IF a < b THEN
    PRINT "a es menor que b"
    PRINT "Esto es otra sentencia más"
ELSE : REM si no...
    PRINT "a no es menor que b"
END IF
```

**En Sinclair BASIC teníamos que ingeniárnoslas haciendo algo así:**

```
1000 IF a < b THEN PRINT "a es menor que b": PRINT "Esto es otra sentencia más" : GO TO 1030
1010 REM si no...
1020 PRINT "a no es menor que b"
1030 ...
```

***WHILE***

**Esta sentencia es nueva ZX BASIC y sirve también para hacer bucles. Es más potente que FOR porque el bucle se repite mientras se dé la condición que se indique. Si al empezar el bucle la condición es falsa, entonces no se llega a ejecutar ninguna iteración. Un ejemplo:**

```
WHILE a < 10
    LET a = a + 1
END WHILE
```

**La finalización del bucle tiene que terminarse con END WHILE o con WEND (son equivalentes). Al igual que con FOR, las sentencias EXIT WHILE y CONTINUE WHILE también pueden utilizarse con WHILE para terminar el bucle o continuar con la siguiente iteración.**

***DO ... UNTIL***

**Similar a la anterior, pero aquí la comprobación de la condición se hace al final del bucle y éste se repite mientras no se cumpla la misma (es decir, mientras sea falsa). Al contrario que con WHILE, el bucle se ejecutará al menos una vez. Un ejemplo:**

```
DO
    LET a = a + 1
LOOP UNTIL a >= 10
```

**Igualmente podemos usar CONTINUE DO y EXIT DO para adelantar el bucle o terminarlo anticipadamente.**

***DO ... WHILE***

**Existe también la construcción DO ... WHILE, idéntica a la anterior, solo que esta repite el bucle mientras la condición se cumpla.**

**Manejo de Memoria**

***PEEK y POKE***

**Son idénticas al Sinclair BASIC, pero ahora, además, está extendidas. Tanto PEEK como POKE admiten especificar el tipo de dato que se guarda en memoria. Por defecto será de tipo byte sin signo (como en el ZX Spectrum). Así pues, las siguientes dos líneas son equivalentes:**

```
LET a = PEEK 16384
LET a = PEEK(uByte, 16384)
```

**Pero a veces queremos guardar o leer un entero de 16 bits. En el mismo manual del ZX Spectrum viene un ejemplo. Estos valores, en el Z80 se leen de esta manera:**

```
REM Guardamos en la variable 'a' la dirección de comienzo de los GDU
LET a = PEEK 23675 + 256 * PEEK 23676 : REM Forma tradicional
LET a = PEEK(Uinteger, 16384)
```

**Ambas formas son equivalentes, pero la segunda es más eficiente (por el ensamblador generado) y más legible. Si se usa la segunda forma, los paréntesis son obligatorios. Igualmente, podemos guardar con POKE valores de más de un byte:**

```
REM Cambiamos la dirección de los GDU según la variable 'a'

REM Forma tradicional
POKE 23675, a - INT(a / 256) * a : REM cuidado con el redondeo si a es entera.
POKE 23676, INT(a / 256)

REM Forma moderna
POKE Uinteger 23675, a
```

**Claramente, la segunda forma es más legible (y preferible). Además, se traduce de forma más eficiente a ensablador. ¡Ahora ya es posible (y extraño), guardar números flotantes con POKE en memoria! Prueba a hacer POKE Float 16384, PI.**

**Salida por Pantalla**

***PRINT***

**Se ha intentado que PRINT sea lo más compatible posible con el original. De hecho, a nivel sintáctico funciona igual. E incluso los códigos de color de la ROM se pueden utilizar. Esta implementación de PRINT no usa la rutina de la ROM (para mayor velocidad) sino que es propia, y permite imprimir en todas las filas de pantalla. Así pues, PRINT AT 22,0; es una sentencia legal. No existen canales.**

***BORDER, PAPER, INK, INVERSE, BRIGHT, FLASH, OVER***

**Funcionan igual que en Sinclar BASIC. Para BORDER, usar un color mayor que 7 se suele ignorar, ya que sólo se toman los 3 bits más bajos. Para INK y PAPER si se puede usar el valor 8 (transparencia). Over tiene funcionalidades extra: OVER 1 actúa como en el ZX Spectrum e imprime realizando la operacion XOR. Pero se puede usar también OVER 2 y OVER 3 en conjunción con el comando PRINT. OVER 2 realiza un AND y OVER 3 realiza un OR. Esto se puede usar para crear efectos de filmation. Prueba el siguiente ejemplo de BORDER y compara su velocidad con la del Basic de la ROM:**

```
10 BORDER 0: BORDER 1: GOTO 10
```

***PLOT, DRAW y CIRCLE***

**Funcionan igual que en el Basic original... pero más rápido. PLOT usa la rutina de la ROM, por lo que se aplican todos atributos de color que usa el comando PLOT original. La diferencia ahora es que disponemos de los 192 puntos de pantalla para pintar. La coordenada (0, 0) es ahora la esquina inferior izquierda física de la pantalla. Eso significa que si usamos un comando de dibujado cualquiera, nuestros dibujos aparecerán 16 píxeles más abajo que en el Sinclair BASIC original, pues ahora disponemos de 16 líneas de altura más.**

**DRAW y CIRCLE están optimizadas (no son las de la ROM) y emplean el algoritmo de Bresenham, por lo que son algo más rápidas (especialmente CIRCLE). También se pueden dibujar arcos, con DRAW x, y, a como en el BASIC original. La rutina está copiada de la ROM, para simular el mismo comportamiento que la original, pero modificada para dibujar también en toda la pantalla, como las anteriores. El siguiente ejemplo está sacado del manual del ZX Spectrum (el reloj), pero está escrito con la nueva sintaxis, sin usar números de línea:**

```
REM Del manual de ZX Spectrum 48K
REM Un programa de Reloj
```

```
REM Primero dibujamos la esfera
CLS
FOR n = 1 to 12
    PRINT AT 10 - (10 * COS(n * PI / 6) - 0.5), 16 + (0.5 + 10 * SIN(n * PI / 6)); n
NEXT n

REM Lo siguiente sería PRINT #0; en el Basic de la ROM
PRINT AT 23, 0; "PULSA UNA TECLA PARA TERMINAR";

FUNCTION t AS ULONG
    RETURN INT((65536 * PEEK (23674) + 256 * PEEK(23673) + PEEK (23672))/50)
END FUNCTION

DIM t1 as FLOAT

OVER 1
WHILE INKEY$ = ""
    LET t1 = t()
    LET a = t1 / 30 * PI: REM a es el ángulo del segundero en radianes
    LET sx = 72 * SIN a : LET sy = 72 * COS a
    PLOT 131, 107: DRAW sx, sy

    LET t2 = t()
    WHILE (t2 <= t1) AND (INKEY$ = "")
        let t2 = t()
    END WHILE : REM Espera hasta el momento de moverlo

    PLOT 131, 107: DRAW sx, sy
END WHILE
```

***SCREEN$, ATTR, POINT***

**Existen y se comportan como en Sinclair BASIC. Pero son funciones externas. Las funciones externas son aquellas que existen en un fichero .BAS aparte. En concreto, en el directorio library/ del compilador hay una biblioteca de funciones que irá creciendo con el tiempo. Para usarlas en tu programa, tienes que usar una directiva de preprocesador como la que sigue:**

```
#include <screen.bas>
```

**Los ficheros que las contienen son SCREEN.BAS, ATTR.BAS y POINT.BAS respectivamente. Si miras en ese directorio, verás que hay otras funciones. Enseguida veremos cómo definir nuestras propias funciones.**

**Sonido**

***BEEP***

**Por ahora, el único soporte de sonido es el comando BEEP, que usa la rutina de la ROM y es idéntico al de Sinclair BASIC. La única ventaja es que aquí, al disponer de mayor rapidez, podemos implementar algunos trucos de sonido. Existe un comando en la versión 128K del BASIC, PLAY que se espera poder implementar en futuras versiones.**

**El siguiente ejemplo está sacado del manual el ZX Spectrum (Frere Gustav del manual de ZX Spectrum 48K, capítulo 19):**

```
10 PRINT "Frere Gustav"
20 BEEP 1,0: BEEP 1,2: BEEP .5,3: BEEP.5,2: BEEP 1,0
30 BEEP 1,0: BEEP 1,2: BEEP .5,3: BEEP.5,2: BEEP 1,0
40 BEEP 1,3: BEEP 1,5: BEEP 2,7
50 BEEP 1,3: BEEP 1,5: BEEP 2,7
60 BEEP .75,7: BEEP .25,8: BEEP .5,7: BEEP .5,5:BEEP .5,3:
       BEEP.5,2: BEEP 1,0
70 BEEP .75,7: BEEP .25,8: BEEP .5,7: BEEP .5,5: BEEP .5,3: BEEP .5,2:
```

```
   BEEP 1,0
80 BEEP 1,0: BEEP 1,-5: BEEP 2,0
90 BEEP 1,0: BEEP 1,-5: BEEP 2,0
```

**Funciones Matemáticas y Números aleatorios**

***STR$, VAL, PI, SIN, COS, TAN, ASN, ACS, ATN, EXP***

**Estas funciones trabajan todas igual que en el BASIC original del ZX Spectrum a excepción de VAL. Como ya se dijo antes, VAL es un caso particular prácticamente imposible de compilar. Funciona como en la mayoría de los BASIC estándar: se convierte una cadena alfanumérica a punto flotante, pero esta cadena sólo puede contener un número (no una expresión). Si una cadena no se puede convertir, se devuelve 0. Luego VAL "x+x" devolverá 0 siempre, aunque la variable x esté definida.**

***RANDOMIZE, RND***

**Existen y se usan como en el ZX Spectrum con la diferencia de que la rutina de números aleatorios es más rápida ya que no usa la calculadora de punto flotante de la ROM sino registros y desplazamientos en ensamblador. Esta rutina es un generador lineal congruente (como la del ZX Spectrum) pero usa números de 32 bits, por lo que tiene un periodo de miles de millones (antes de que vuelva a repetirse la secuencia). Se ha sacado del libro Numerical Recipes in C (disponible gratuitamente en internet, aunque vale la pena comprarlo). Además, tiene una mejor dispersión. Prueba el siguiente programa en el ZX Spectrum, tanto en el BASIC de la ROM como en ZX BASIC (compilado):**

```
10 LET x = INT(RND * 256): LET y = INT(RND * 175)
20 PLOT x, y
30 GOTO 10
```

**Aparte de la velocidad del programa compilado, notarás que en la versión del BASIC de Sinclair aparecen líneas diagonales. Ello indica que la aleatoriedad de los números no es tan alta en el BASIC de la ROM como se podría esperar.**

**Entrada y salida**

***INKEY$, IN, OUT***

**También están presentes y funcionan de forma idéntica a la de Sinclair BASIC... excepto por su velocidad, que es bastante mayor en el caso de IN y OUT.**

**Caracteres gráficos, GDU y códigos de color**

**Para poder introducir caracteres gráficos y códigos de color, se ha seguido el mismo convenio que BASIN (un entorno para programar y depurar programas BASIC del ZX Spectrum). Así, dentro de una cadena de caracteres, el caracter de barra invertida '\' tiene un significado especial. Así, para escribir un el GDU "A", escribiremos PRINT "\A". Si queremos escribir varios carácteres gráficos seguidos, "AB", escribiremos: PRINT "\A\B". Si queremos imprimir la barra invertida (este carácter existe en el Spectrum), usaremos doble barra: PRINT "\\"**

**Los símbolos gráficos (aquellos que tenían formas de cuadraditos y que se sacaban en con el cursor en modo gráfico y pulsando un número del 0 al 8) también pueden ponerse como en BASIN. Prueba a poner PRINT "\:.\'.". Como puedes ver, se componen de la barra invertida seguido de dos caracteres que pueden ser uno de estos: [ ][.][']][:] (espacio en blanco, punto, apóstrofe y dos puntos). Cada "puntito" del caracter representa un cuadrado del gráfico (es difícil de explicarlo, lo mejor es que lo pruebes). Estos códigos son leídos por el compilador y reemplazados por un solo byte, correspondiente al código ASCII del caracter que queramos representar. Puedes probar a hacer un programa BASIC en Basin y grabarlo como .bas (ASCII) y ver la secuencia de caracteres generada.**

**Igualmente, los códigos de color, brillo, flash e inversión de vídeo pueden especificarse "en línea", como se hacía con el ZX Spectrum, siguiendo el mismo convenio que BASIN. Para escribir por ejemplo, HOLA con tinta roja sobre fondo negro, puedes hacer: PRINT "{i2}{p0}HOLA"**

**Los códigos son:**

| Código | Significa | Valores |
|---|---|---|
| {iN} | Tinta | N = 0..7 |
| {pN} | Papel | N = 0..7 |
| {fN} | Flash | N = 0..1 |
| {vN} | Inverse | N = 0..1 |
| {bN} | Brillo | N = 0..1 |

Además, puedes especificar cualquier caracter ASCII en decimal, usando \#xxx. Por ejemplo, el copyright puedes ponerlo como PRINT "*" o bien como PRINT "\#127"

## Funciones y Subrutinas

Una de las grandes diferencias del ZX BASIC respecto a sinclair BASIC es la posibilidad de definir funciones y subrutinas, y que además estas contengan variables privadas. La única posibilidad que ofrecía el BASIC de Sinclair para definir funciones era con DEF FN, que aquí ya no existe. Para definir una función, usaremos la palabra reservada FUNCTION, así:

```
FUNCTION mifuncion(x AS Integer, y AS Byte) AS Integer
    REM Función que devuelve x + y
    RETURN x + y
END FUNCTION
```

Esta pequeña función recibe 2 parámetros, un entero de 16 bit en x y un byte en y, para devolver la suma de ambos (x + y).

Para usar la función, sólo tienes que llamarla como como si fuera una función BASIC cualquiera:

```
PRINT mifuncion(3, 5) : REM Imprime 8
```

Mira el ejemplo anterior del reloj, cómo define y usa una función.

También se puede llamar a una función sin más, como si fuera una subrutina:

```
mifuncion(3, 5) : REM suma 3 + 5 y luego los descarta y no hace nada con el resultado
```

Al igual que las funciones, también puedes definir subrutinas. Se definen usando la palabra reservada SUB. La sintaxis es muy similar a la anterior:

```
SUB misubrutina(x AS Integer, y AS Byte)
    REM Función que devuelve x + y
    PRINT x + y
END SUB
```

Y la invocamos así

```
misubrutina(3, 6) : REM imprime 9
```

La diferencia principal es que SUB siempre tiene que invocarse. Nunca devuelve un resultado, al contrario que las funciones. De hecho, no puedes usar RETURN <valor> para salir de una subrutina, sino simplemente RETURN. Retornar de una subrutina o función y retornar de un GOSUB son cosas distintas. Al escribir simplemente RETURN, desde dentro de una función o subrutina siempre se supondrá que se retorna de la misma y no de un GOSUB.

Esto es así porque cada vez que se retorna, el programa tiene que hacer ciertas gestiones con la pila de código máquina antes de regresar. Así pues, no es lo mismo retornar de un GOSUB, que de una función o subrutina. Al declarar una subrutina ya le estamos indicando al compilador que no es necesario devolver ningún valor, y eso permite hacer una pequeña optimización.

Por último, las variables declaradas (DIM) dentro de una subrutina o función son privadas y no son accesibles desde fuera. Además, sólo usan memoria durante la ejecución de la función. Al salir de la misma, esa memoria se libera (se usa la pila de código máquina). Mira este ejemplo:

```
SUB Ejemplo
    DIM a as Integer: REM esta variable es privada

    LET a = 5
    PRINT "En la subrutina, a="; a
END SUB
```

```
LET a = 10
PRINT "Antes de la subrutina, a="; a
Ejemplo() :  REM  llamamos a la subrutina ejemplo e imprime la variable privada
PRINT "Despues de la subrutina, a ="; a
```

**ASM integrado**

**Es posible meter líneas de ASM directamente. Esto es una característica de muy bajo nivel, y se sale un poco de los propósitos de este artículo. Básicamente, se puede incrustar ASM directamente en el código así:**

```
ASM
...
...
END ASM
```

**Como ZX Basic aspira a ser multiplataforma (reorientable), el código que hagas con ASM directo sólo compilará en la arquitectura que soporte ese ensamblador (en nuestro caso, Z80). Aquí tienes un ejemplo:**

```
ASM
ld a, 0FFh
ld (16384), a
END ASM
```

**La idea de usar ASM directamente es en aquellos bucles que requieran mucha velocidad (por ejemplo, alguna subrutina de algún juego). Si miras la biblioteca de funciones, library/ verás que muchas funciones están definidas en ensamblador.**

**Lo que falta...**

**No todo iba a ser perfecto. ZX Basic, por ser compilado, tiene cosas que le faltan. Por ejemplo, no existe INPUT. Te la tendrás que implementar (la mayoría de los juegos se implementan su propio INPUT; fíjate en El Hobbit). Ya hay una función INPUT implementada en en la biblioteca de funciones, por si te interesa esa. Prueba a usarla.**

```
REM Importamos la función input
#include <input.bas>

LET a$ = input(32): REM Input de una cadena de 32 caracteres como máximo
PRINT a$
```

**Es muy difícil utilizar el INPUT de la ROM, que usa canales (que ya no existen aquí), y otras zonas del área de BASIC que pueden corromper el programa compilado o la pila de ejecución (aunque se supone que el CLEAR del cargador evitará esto). Tampoco tienen sentido READ, DATA y RESTORE. Son un desperdicio de memoria e imposibles de compilar tal y como las ofrece el BASIC de la ROM; aunque es probable que se implementen en un futuro con ciertas limitaciones.**

**Ejemplos**

**Este artículo está quedando muy extenso, así que lo cortaré aquí (si fuera para MicroHobby, daría para varios números). Quedan algunas cosas por explicar (como el alias de variables o las instrucciones de manejo de bits: rotaciones, etc.) En cuaquier caso, en el directorio examples tienes varios ejemplos que puedes compilar para aprender cómo funcionan. El ejemplo de la Bandera Inglesa está sacado del manual del ZX Spectrum y funciona tal cual está, sin ninguna modificación. Otros ejemplos, como el Snake (cortesía de Federico J. Alvarez Valero, año 2003) se han retocado para declarar explícitamente el tipo de dato de algunas variables (recordemos que trabajar con enteros es más rápido que punto flotante), y terninar los IF con sus END IF correspondientes.**

**Y pasaron los años...**

**¿20 años? ¡Glup! Al menos se cumplió una parte del sueño (la otra no os la digo...). ¿Fue demasiado tarde? No lo creo. Al menos no en parte.**

**El ZX Spectrum, los micros en general, y su época de alguna manera dejaron huella. Muchos pensamos que esto fue irrepetible. En aquella época la tecnología no avanzaba tan rápidamente como ahora y es por eso que micros como el ZX Spectrum duraron de media una década sin sufrir grandes cambios. Por contra las**

videoconsolas y sistemas actuales apenas pasan de los 3 ó 4 años de vida.

Hoy día estamos tan saturados y acostumbrados a los cambios que ya nada nos asombra (o al menos a mí). Miles de millones de píxeles en miles de millones de colores con un sonido envolvente no consiguen dejar esa impronta que una máquina tan simple logró en apenas unas horas. Quizá porque fue la primera, no lo sé. Pero me consta que las generaciones actuales no sienten esa pasión y ese tirón por las máquinas. La PS3, la XBOX, el iPhone o el PC. Ninguno consigue enganchar tanto a los chicos de hoy día.

Suena un poco nostálgico, lo sé, pero creo que hemos tenido la suerte de haberlo podido vivir.

**Enlaces y ficheros**

- Ejemplos en BASIC listos para compilar con ZXBasic.

Boriel







# Z88DK v1.8 y SP1 (SPLIB3)

*Desde la última entrega de nuestro curso de Z88DK ha llovido mucho, y hay bastantes novedades que contar. La principal de ellas es que el 9 de Marzo del 2008 se liberó la versión 1.8 de Z88DK.*

Una de las novedades más destacables de la versión 1.7 fue la integración de SPLIB en z88dk, rebautizando a splib3 como SP1. En nuestro anterior artículo comenzamos a utilizar SP1 a través de ejemplos que mostraban la instalación del compilador y la utilización de esta librería de funciones para juegos.

En este artículo mostraremos los diferentes módulos de SP1 y otras bibliotecas interesantes de Z88DK para su utilización en la creación de programas y juegos.

## La nueva librería SP1

SP1 es una librería de sprites por software (puesto que el Spectrum no dispone de chip hardware para el tratamiento de Sprites), que como el lector sabe, está diseñada para minimizar la cantidad de dibujados de bloques gráficos en pantalla.

SP1 es la nueva versión de la antigua librería SPLIB. En esta versión (SPLIB3), la biblioteca se divide en diferentes módulos para que el programador pueda elegir cuáles de ellos desea usar y cuales no. Hasta ahora, si un programador quería usar las funciones de teclado o de modo IM2 de interrupciones de SPLIB, se veía obligado a incluir la biblioteca completa, con la consiguiente pérdida de memoria, espacio "ejecutable" y tiempo de carga. Ahora, es posible incluir o no cada módulo individualmente.

## Librerías de Z88DK

A continuación se detallan algunas de las librerías incluídas con Z88DK. Algunas son parte del compilador desde sus inicios, y otras son, como ya hemos comentado, diferentes módulos de la antigua SPLIB2, ahora integrada con el nombre de SP1.

**Tipos Abstractos de Datos**

- Fichero de cabecera a incluir: #include <atd.h>
- Librería a enlazar: -ladt.

Esta librería incluye funciones genéricas para algunos tipos abstractos de datos, como Listas Enlazadas (Linked Lists), Pilas (Stacks), y Colas (Queues). Por el momento, estos tipos de datos son dinámicos (se reserva y libera memoria con su uso), aunque están previstas también implementaciones estáticas. Otros tipos de datos (árboles, buffers circulares, etc.) serán implementados en el futuro.

Veamos algunas funciones de ejemplo (en este caso, de Pilas / Stacks) que muestran "el aspecto" de la librería:

```
adt_StackCreate();
adt_StackDelete();
adt_StackPush();
adt_StackPop();
adt_StackPeek();
adt_StackCount();
```

La Página de documentación de libadt muestra detallados ejemplos de uso de la librería.

**Peticiones de memoria**

- Fichero de cabecera a incluir: #include <malloc.h> e #include <balloc.h>
- Librería a enlazar: -lmalloc y -lballoc.

Esta librería incluye funciones para pedir y liberar memoria dinámicamente, en lugar de utilizar arrays vacíos

estáticos (que aumentan el tiempo de carga).

Veamos algunas funciones de ejemplo:

```
mallinit();
calloc();
malloc();
free();
realloc();
```

La Página de documentación de malloc y balloc muestra detallados ejemplos de uso de la librería.

**Modo 2 de Interrupciones**

- Fichero de cabecera a incluir: #include <im2.h>
- Librería a enlazar: -lim2.

Esta librería incluye funciones de gestión del modo 2 de Interrupciones. Estas funciones nos permitirán enlazar la ejecución de código con el modo 2 de interrupciones, para así poder implementar temporizadores y funciones sincronizadas con IM2.

Algunas de las funciones de que provee esta librería son:

```
im2_init();
im2_InstallISR();
im2_EmptyISR();
im2_CreateGenericISR();
```

La Página de documentación de im2.h muestra información detallada sobre los modos de interrupciones y ejemplos de uso de la librería.

**Teclado, joystick y ratón**

- Fichero de cabecera a incluir: #include <input.h>.
- Librería a enlazar: No es necesaria.

La biblioteca input.h incluye las funciones necesarias para la lectura del teclado en nuestros juegos y programas. Algunas de las funciones más útiles dentro de esta librería son las siguientes:

```
in_GetKeyReset();
in_GetKey();
in_InKey();
in_LookupKey();
in_KeyPressed();
in_WaitForKey();
in_WaitForNoKey();
in_Pause();
in_Wait();
in_JoyKeyboard();
```

Por otra parte, input.h también contiene todas las funciones necesarias para acceder a ratón y joystick, con funciones como las siguientes. Concretamente, la anterior función in_JoyKeyboard() emula la lectura de los joysticks con el mismo formato que la lectura de teclas, con in_LookupKey(). Para hacer uso de la emulación de joystick, será necesario incluir el fichero de cabecera <spectrum.h>.

Como muestra, el siguiente ejemplo tomado de la documentación de la librería input, que combina el uso de in_JoyKeyboard() con las llamadas a in_LookupKey():

```
#include <input.h>
#include <spectrum.h>

// example for ZX Spectrum target which supplies
// the following device specific joystick functions:
// in_JoyKempston(), in_JoySinclair1(), in_JoySinclair2()

uchar choice, dirs;
void *joyfunc;                // pointer to joystick function
```

```
char *joynames[] = {            // an array of joystick names
   "Keyboard QAOPM",
   "Kempston",
   "Sinclair 1",
   "Sinclair 2"
};
struct in_UDK k;

// initialize the struct_in_UDK with keys for use with the keyboard joystick

k.fire  = in_LookupKey('m');
k.left  = in_LookupKey('o');
k.right = in_LookupKey('p');
k.up    = in_LookupKey('q');
k.down  = in_LookupKey('a');

// print menu and get user to select a joystick

printf("You have selected the %s joystick\n", joynames[choice]);
switch (choice) {
   case 0 : joyfunc = in_JoyKeyboard; break;
   case 1 : joyfunc = in_JoyKempston; break;
   case 2 : joyfunc = in_JoySinclair1; break;
   default: joyfunc = in_JoySinclair2; break;
}

...

// read the joystick through the function pointer

dirs = (joyfunc)(&k);
if (dirs & in_FIRE)
   printf("pressed fire!\n");

...
```

**Librería de Sonido**

- Fichero de cabecera a incluir: #include <sound.h>.
- Librería a enlazar: No es necesaria.

Esta librería de sonido contiene funciones muy básicas de reproducción sonora, así como algunos "efectos especiales" estándar.

```
bit_open();
bit_close();
bit_click();
bit_fx(N);
bit_fx2(N);
bit_fx3(N);
bit_fx4(N);
bit_synth();
bit_beep();
bit_frequency();
bit_play();
```

Más información sobre sound.h en la página oficial de la librería.

**Librería de Sprites SP1**

- Fichero de cabecera a incluir: #include <sprites/sp1.h>.
- Librería a enlazar: -lsp1.

La librería sprites/SP1 contiene todo el código dedicado a Sprites de la antigua SPLIB: creación de sprites, movimiento, borrado, etc.

Algunas de las funciones que se detallan a continuación ya fueron mostradas en nuestro anterior entrega, incluyendo ejemplos de uso:

```
sp1_CreateSpr();
sp1_AddColSpr();
sp1_DeleteSpr();
sp1_MoveSprAbs();
sp1_MoveSprRel();
sp1_TileEntry();
sp1_PrintAt();
sp1_GetTiles();
sp1_PutTiles();
sp1_ClearRect();
```

Más información sobre SP1 en la página de SP1 en el wiki de Z88DK.

Esta biblioteca debe compilarse desde los fuentes de Z88DK para poder utilizarla. Esto se hace entrando en el directorio sp1 y ejecutando el comando make sp1-spectrum:

```
$ cd $Z88DK
$ cd libsrc/sprites/software/sp1/
$ make sp1-spectrum
```

**Librería general "spectrum.h"**

- Fichero de cabecera a incluir: #include <spectrum.h>.
- Librería a enlazar: No es necesaria.

La librería spectrum.h contiene funciones varias, como acceso directo a Joystick y Ratón, identificar el modelo de Spectrum que ejecuta nuestro programa, detectar la existencia de periféricos conectados al Spectrum, funciones de cinta (save/load), cambio de color del borde, cálculo de direcciones de atributos, etc.

A continuación se muestran algunas de las funciones que podremos utilizar mediante la inclusión de spectrum.h:

```
zx_type();
zx_model();
zx_soundchip();
zx_kempston();
zx_basemem();
tape_save();
tape_load_block();
tape_save_block();
in_JoyFuller();
in_JoyKempston();
in_JoySinclair1();
in_JoySinclair2();
in_MouseAMXInit();
in_MouseAMX();
zx_border();
zx_attr();
zx_screenstr();
x_cyx2saddr(); (y derivados)
```

La biblioteca define también gran variedad de constantes para su uso en nuestros programas, como colores (BLACK, BLUE, RED, MAGENTA, etc.), identificadores de los tokens del BASIC, etc.

Se puede consultar la página de spectrum.h para más información.

**Otras librerías**

Otras bibliotecas de las que podemos hacer uso:

- Funciones de E/S: stdio.h.
- Funciones de reloj y tiempo: time.h.
- Funciones de acceso al puerto serie: rs232.h.
- Implementación del algoritmo A*: algorithm.h.
- Funciones de cadena: string.h.

Puede encontrarse la documentación de cada una de ellas en la página principal de Z88DK.

# Integrando ASM de Z80 en Z88DK

Una de las cosas más interesantes de Z88DK es que nos permite utilizar ensamblador en-línea dentro de nuestro código en C. Gracias a esto, podemos identificar las rutinas más críticas en necesidades de velocidad (por ejemplo, rutinas gráficas, de sonido, etc), y reescribirlas en ensamblador si es necesario.

Esto permite acelerar el ciclo de desarrollo y depuración, ya que es posible inicialmente escribir el programa o juego íntegramente en C (simplificando mucho la estructura general del programa), para después pasar a convertir en ensamblador, una a una, aquellas rutinas importantes y críticas.

El código en lenguaje ensamblador se inyecta dentro de nuestro código C con la directiva asm.

```
asm("halt");
```

Para incluir más de una línea de código ensamblador consecutiva, se recurre a las directivas #asm y #endasm. El siguiente código vacía el contenido de la pantalla:

```
#asm
   ld hl, 16384
   ld a, 0
   ld (hl), a
   ld de, 16385
   ld bc, 6911
   ldir
#endasm
```

La principal utilidad será, habitualmente implementar el código ensamblador en funciones C para llamarlas desde otras partes de nuestro programa:

```
void BORDER_BLACK( void )
{
  #asm
     ld c, 254
     out (c), a
     ld hl, 23624
     ld a, 0
     ld (hl), a
  #endasm
}
```

Pero para poder aprovechar la integración entre C y ASM, necesitamos conocer los siguientes mecanismos:

- Creación y referencia de etiquetas ASM.
- Acceso a variables de C desde bloques ASM.
- Definición de variables en ASM utilizables desde C.
- Lectura desde bloques ASM de los parámetros pasados a las funciones.
- Devolución de valores desde bloques ASM llamados como funciones.

A continuación se detalla la forma de realizar esto desde Z88DK.

**Creación y referencia de etiquetas ASM**

Si tenemos que realizar rutinas medianamente largas, necesitaremos utilizar etiquetas para las estructuras condicionales. A continuación se muestra un ejemplo de declaración de una etiqueta y la referencia a la misma:

```
#asm
   LD B, 8

.mirutina_loop               ; la etiqueta lleva punto delante

   LD A, (HL)
   (...)
   LD C, A
   AND 7
   JR Z, mirutina_loop       ; la referencia a la etiqueta, no.
```

```
#endasm
```

Como puede verse, las etiquetas se definen con un punto delante y se referencian sin el punto.

Nótese que hemos utilizado el nombre de la rutina dentro de la etiqueta. Esto es así porque las etiquetas son 2 globales y no podemos llamar a 2 etiquetas de igual forma en 2 funciones diferentes. Por eso, en vez de "loop", hemos usado "mirutina_loop", de forma que en una segunda rutina usaremos "mirutina2_loop".

**Accediendo a variables de C desde ASM**

Dentro de los bloques de ASM podemos acceder a las variables globales definidas en el código de C. Esto se mediante el símbolo de subrayado antes del nombre de la variable:

```
char borde;

void BORDER( void )
{
#asm
   (...)
   // Leer variable:
   ld hl, 23624
   ld a, (_borde)
   ld (hl), a

   // Escribir variable:
   ld (_borde), a

   // Escribir variable (forma 2):
   ld hl, _borde
   ld a, (hl)

#endasm
}
```

Es de vital importancia, a la hora de leer y escribir valores en variables, que respetemos el tipo de variable en que estamos escribiendo. Si estamos tratando de modificar una variable de 1 sólo byte (apuntada por HL), utilizaremos una operación del tipo LD (HL), A para que, efectivamente, se escriba un sólo byte en la posición de memoria apuntada por la variable. Si escribimos más de un byte, estaremos afectando al byte siguiente al de la variable en cuestión, con lo que modificaremos el valor de la siguiente variable o bloque de código en memoria.

Recuerda para esto que:

- signed y unsigned char -> 1 byte.
- signed y unsigned short -> 2 bytes.
- signed y unsigned int -> 2 bytes.
- signed y unsigned char * -> 2 bytes (es una dirección de memoria).

Finalmente, cabe destacar que sólo se pueden acceder a variables globales: no se podrá acceder desde esta forma a variables definidas dentro de las funciones, puesto que son locales y están localizadas en la pila. Tampoco se podrá acceder de esta forma a los parámetros pasados a las funciones.

Es decir, los siguientes ejemplos no son válidos:

```
void BORDER( unsigned char borde )
{
   char varlocal;

#asm
   ld a, (_varlocal)  // MAL
   (...)
   ld hl, 23624
   ld a, (_borde)     // MAL
   ld (hl), a
#endasm
}
```

**Uso de variables temporales desde ASM**

También es posible definir "variables" y bloques de datos dentro de lo que es el "binario ejecutable" del programa, y refenciar a ellos después desde ensamblador. Esto es habitual a la hora de utilizar "variables temporales" o de almacenamiento en rutinas ensamblador cuando tenemos necesidades de almacenamiento que hacen a los registros insuficientes.

Como no vamos a referenciar las variables desde C, no tenemos que utilizar el carácter de subrayado antes del nombre de la variable.

Recuerda, a la hora de guardar los datos dentro de las etiquetas que hacen referencia a ellas, que debes guardar el número de bytes adecuado para no machacar datos o código que vaya tras ellos:

```
#asm
   ld a, (hl)
   ld (valor_temp), a     ; guardamos un byte (A)
   inc hl

   ld c, (hl)
   inc hl
   ld b, (hl)
   inc hl                 ; construimos BC como un word
   ld (valor_int), bc     ; guardamos un word (2 bytes)

   (...)

   ret

 valor_temp  defb  0
 valor_int   defw  0
#endasm
```

**Definición de variables en bloques ASM**

Acabamos de ver cómo acceder en los bloques de ASM a variables definidas desde C. A continuación veremos el proceso inverso: definiremos variables (incluso bloques de datos contiguos) dentro de bloques de ASM que después podrán ser referenciadas desde código C.

Para ello, definimos al principio del código C las variables tal y como las referenciaremos desde C. Es importante indicar el tipo de dato adecuado (char, int, o char *):

```
extern unsigned char caracter;
extern unsigned int dato_int;
extern unsigned char sprite0[];
```

Después, en otra zona del programa (normalmente al final del código o en un fichero .c/.h aparte), se define el dato o array de datos al que referencian las variables que hemos indicado como extern. Para ello utilizaremos las directivas DEFB, teniendo en cuenta que el tipo de la variable C nos indica la longitud que debe tener el dato.

```
#asm
._caracter
        DEFB 52

._dato_int
        DEFB 0, 0

._sprite0
        DEFB 199,56,131,124,199,56,239,16
        DEFB 131,124,109,146,215,40,215,40
        (...etc...)
#endasm
```

En realidad, dentro del bloque #asm sólo estamos definiendo etiquetas a direcciones de memoria (igual que pueda ser la de un bucle) que coincidirán, para el compilador, con las variables (que también son etiquetas a direcciones) definidas como externs del mismo nombre.

**Pasando parámetros a funciones ASM**

Hemos visto que no es posible acceder directamente desde los bloques #asm a las variables locales de una función, así como a los parámetros de llamada, ya que en realidad están almacenados en la pila. Pero aunque no se pueda acceder de forma directa a estos valores, sí que podemos acceder a ellos leyéndolos mediante la ayuda del puntero de pila SP.

Sin utilizar la pila, la mejor forma de pasar parámetros a las funciones es usar algún conjunto de variables globales que utilicemos en las funciones:

```
// Definimos unas cuantas variables globales que usaremos
// como parametros para nuestras funciones que usen ASM:

char auxchar1, auxchar2, auxchar3, auxchar4;
int  auxint1, auxint2, auxint3, auxint4;

// A la hora de llamar a una función, lo haríamos así:

auxchar1 = 10;
auxchar2 = 20;
MiFuncion();

// De este modo, la función podria hacer lo siguiente:
void MiFuncion( void )
{
#asm
     ld a, (_auxchar1)
     ld b, a
     ld a, (_auxchar2)
     (etc...)
#endasm
}
```

Aunque este método es fáctible (y rápido), resulta más legible el evitar la utilización de este tipo de variables (que haría complicada la creación de funciones recursivas o anidadas). Para ello, utilizaremos el sistema de paso de argumentos basado en la pila.

En C (y en otros lenguajes de programación) los parámetros se insertan en la pila en el orden en que son leídos. La subrutina debe utilizar el registro SP (mejor dicho, una copia) para acceder a los valores apilados en orden inverso. Estos valores son siempre de 16 bits aunque las variables pasadas sean de 8 bits (en este caso ignoraremos el byte que no contiene datos, el segundo).

Veamos unos ejemplos:

```
  int jugador_x, jugador_y;

  jugador_x = 10;
  jugador_y = 200;
  Funcion( jugador_x, jugador_y );
  (...)


//------------------------------------------------
int Funcion( int x, int y )
{

#asm
   ld hl, 2
   add hl, sp          ; Ahora SP apunta al ultimo parametro metido
                       ; en la pila por el compilador (el valor de Y)
   ld c, (hl)
   inc hl
   ld b, (hl)
   inc hl              ; Ahora BC = y

   ld e, (hl)
   inc hl
   ld d, (hl)
   inc hl              ; Ahora, DE = x
```

```
    ; ahora hacemos lo que queramos en asm
    ; utilizando DE y BC, que son los valores
    ; pasados como X e Y

#endasm
}
```

No tenemos que preocuparnos por hacer PUSH y POP de los registros para preservar su valor dado que C lo hace automáticamente antes y después de cada #asm y #endasm.

El problema es que conforme crece el número de parámetros apilados, es posible que tengamos que hacer malabarismos para almacenarlos, dado que no podemos usar HL (es nuestro puntero a la pila en las lecturas). Veamos el siguiente ejemplo con 3 parámetros, donde tenemos que usar PUSH para guardar el valor de DE y EX DE, HL para acabar asociando el valor final a HL:

```
int Funcion( int x, int y, int z )
{

#asm
   ld hl, 2
   add hl, sp            ; Ahora SP apunta al ultimo parametro metido
                        ; en la pila por el compilador (valor de Z)
   ld c, (hl)
   inc hl
   ld b, (hl)
   inc hl               ; Ahora BC = z

   ld e, (hl)
   inc hl
   ld d, (hl)
   inc hl               ; Ahora, DE = y

   push de              ; Guardamos DE

   ld e, (hl)
   inc hl
   ld d, (hl)
   inc hl               ; Usamos DE para leer el valor de x

   ex de, hl            ; Ahora cambiamos x a HL
   pop de               ; Y recuperamos el valor de y en DE

   ; (ahora hacemos lo que queramos en asm)

#endasm
}
```

La manera de leer bytes (char) pulsados en C es de la misma forma que leemos una palabra de 16 bits, pero ignorando la parte alta. En realidad, como la pila es de 16 bits, el compilador convierte el dato de 8 bits en uno de 16 (rellenando con ceros) y pulsa este valor:

```
int Funcion( char x, y )
{

#asm
   ld hl,2
   add hl,sp            ; Ahora SP apunta al ultimo parametro metido
                        ; en la pila por el compilador (y)

   ld a, (hl)           ; Aquí tenemos nuestro dato de 8 bits (y)
   ld b, a
   inc hl
   inc hl               ; La parte alta del byte no nos interesa

   ld a, (hl)           ; Aquí tenemos nuestro dato de 8 bits (x)
   ld c, a
```

```
   inc hl
   inc hl              ; La parte alta del byte no nos interesa

   ; (ahora hacemos lo que queramos en asm)

#endasm
}
```

En ocasiones, es posible que incluso tengamos que utilizar variables auxiliares de memoria para guardar datos. En este caso, utilizamos memoria adicional pero evitamos el uso de la pila.

```
int Funcion( int x, int y, char z )
{

 #asm
   ld hl, 2
   add hl, sp          ; Ahora SP apunta al ultimo parametro metido
                       ; en la pila por el compilador (Z)

   ld c, (hl)
   inc hl
   ld b, (hl)
   inc hl              ; Ahora BC = y
   ld (valor_y), bc    ; nos lo guardamos, BC libre de nuevo

   ld c, (hl)
   inc hl
   ld b, (hl)
   inc hl
   ld (valor_x), bc    ; Nos lo guardamos, BC libre de nuevo

   ld a, (hl)
   ld (valor_z), a     ; Nos guardamos el byte
   inc hl
   inc hl              ; La parte alta del byte no nos interesa

   (ahora hacemos lo que queramos en asm)
   RET

 valor_x  defw  0
 valor_y  defw  0
 valor_z  defb  0
 #endasm
}
```

**Devolución de valores por funciones**

Por contra, para devolver valores no se utiliza la pila (dado que no podemos tocarla), sino que se utiliza el registro HL. Al finalizar la función C, el valor que contenga el registro HL será el devuelto y asignado en la llamada.

En tal caso, cuando realicemos una llamada a una función que contenga un bloque ASM, el valor indicado en HL en el momento de retorno será el devuelto por la misma.

Supongamos pues la siguiente llamada:

```
valor = MiFuncion( a, b, c);
```

En este caso, a valor se le asignará el contenido del registro HL. Como tipos de variables de devolución podremos usar int, short o char (en este último caso sólo se asignará la parte baja de HL).

A continuación, como ejemplo, se muestra una función que dadas unas coordenadas X, Y de pantalla (con X entre 0 y 31 e Y entre 0 y 24), devuelve la dirección de memoria donde se puede alterar el atributo correspondiente a dichas coordenadas. El ejemplo muestra cómo recibir variables por la pila, realizar cálculos con ellas, y devolver un valor en HL:

```
//
// Devuelve la direccion de memoria del atributo de un caracter
// de pantalla, de coordenadas (x,y). Usando la dirección que
// devuelve esta función (en HL, devuelto en la llamada), podemos
// leer o cambiar los atributos de dicho carácter.
//
// Llamada:   valor =  Get_LOWRES_Attrib_Address( 1, 3 );
//
int Get_LOWRES_Attrib_Address( char x, char y )
{
#asm
  ld hl, 2
   add hl, sp                  ; Leemos x e y de la pila
   ld  d, (hl)  ; d = y
   inc hl                      ; Primero "y" y luego "x".
   inc hl                      ; Como son "char", ignoramos parte alta.
   ld  e, (hl)  ; e = x

   ld h, 0
   ld l, d
   add hl, hl                  ; HL = HL*2
   add hl, hl                  ; HL = HL*4
   add hl, hl                  ; HL = HL*8
   add hl, hl                  ; HL = HL*16
   add hl, hl                  ; HL = HL*32
   ld d, 0
   add hl, de                  ; Ahora HL = (32*y)+x
   ld bc, 16384+6144           ; Ahora BC = offset attrib (0,0)
   add hl, bc                  ; Sumamos y devolvemos en HL
#endasm
}
```

**En resumen**

La intención con este artículo final del curso de Z88DK era cerrar un conjunto de contenidos básicos y necesarios a la hora de desarrollar aplicaciones y juegos con este fantástico compilador cruzado de C. En la anterior entrega vimos su instalación y la utilización de la librería SP1, y en esta se han detallado los diferentes módulos de SP1 y la integración de ASM dentro de nuestros programas en C.

Ambos capítulos, unidos, deberían permitir a un lector con conocimientos de C comenzar su andadura en el desarrollo de aplicaciones o juegos en C para Spectrum, incluyendo la posibilidad de utilizar ensamblador en aquellas partes del programa que lo requieran.

Santiago Romero

En este último número de la revista hacemos una promoción 2x1. Llévese los dos últimos artículos del curso de ensamblador por el precio de uno.

# Lenguaje Ensamblador del Z80 (IV)

**La pila y las llamadas a subrutinas**

Los 2 temas que vamos a tratar hoy, la pila por un lado y las llamadas a subrutinas por otro, están íntimamente relacionados. Como veremos, la pila del Spectrum es aquello que permite la utilización de rutinas a las que pasaremos parámetros, y que volverán al punto donde fueron llamadas tras su ejecución.

**La pila del Spectrum**

Hoy vamos a tratar una de las cosas más importantes del microprocesador Z80: **la pila** o **Stack** (en inglés).

La pila, teóricamente, es *una porción de memoria donde se pueden almacenar valores de 16 bits*.

Su nombre viene del hecho que los datos se almacenan unos encima de otros, como en una pila de platos. Cuando almacenamos un nuevo plato en una pila, lo dejamos encima del todo de la misma, sobre el plato anterior. Cuando queremos coger un plato, cogemos el plato más alto, el que está en la parte más alta de la pila.

Las pilas son una estructura de datos conocida como "estructura LIFO": "Last In, First Out": el último que entró es el primero que sale. En nuestro ejemplo de los platos, efectivamente cuando retiramos un plato (el primero que sale) extraemos el que está arriba del todo (el último que habíamos dejado). En una pila de ordenador (como en nuestra pila de datos) sólo podemos trabajar con el dato (o plato) que está arriba del todo de la pila: no podemos extraer uno de los platos intermedios. Sólo podemos apilar un plato nuevo y desapilar el plato apilado arriba del todo de la pila.

La pila del Spectrum no es de platos sino de valores numéricos de 16 bits. Introducimos valores y sacamos valores mediante 2 instrucciones concretas: **PUSH <valor>** y **POP <valor>**, donde normalmente <valor> será un registro (metemos en la pila el valor que contiene un registro de 16 bits, o bien leemos de la pila un valor y lo asignamos a un registro de 16 bits).

Por ejemplo, podemos guardar el valor que contiene un registro en la pila si tenemos que hacer operaciones con ese registro para así luego recuperarlo tras realizar la tarea que tengamos que realizar:

```
LD BC, 1000
PUSH BC         ; Guardamos el contenido de BC en la pila

LD BC, 2000
(...)           ; Operamos con BC

LD HL, 0
ADD HL, BC      ; y ya podemos guardar el resultado de la operación
                ; (recordemos que no existe "LD HL, BC", de modo que
                ; lo almacenamos como HL = 0+BC

POP BC          ; Hemos terminado de trabajar con BC, ahora
                ; recuperamos el valor que tenia BC=1000.
```

La instrucción "PUSH BC" lo que ha hecho es introducir en memoria, en lo alto de la pila, el valor 1000, que recuperamos posteriormente con el "POP BC".

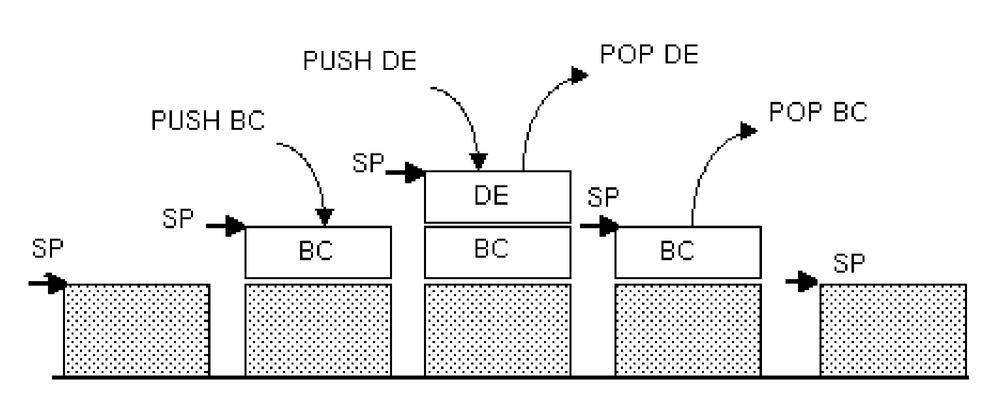
La realidad es que *el Spectrum no tiene una zona de memoria especial o aislada de la RAM dedicada a la pila. En su lugar se utiliza la misma RAM del Spectrum (0-65535).*

El Z80 tiene un registro conocido como **SP (Stack Pointer)**, o **puntero de pila**, que es un registro de 16 bits que contiene una dirección de memoria. Esa dirección de memoria es *"la cabeza de la pila"*: apunta al próximo

lugar donde almacenaremos un dato (al hueco donde guardaremos el próximo plato).

La peculiaridad de la pila del Spectrum es que *crece hacia abajo*, en lugar de hacia arriba. Veamos un ejemplo práctico:




*Como crece y decrece la pila del Z80*

Supongamos que SP (puntero de pila) apunta a 65535 (la última posición de la memoria). Hagamos una serie de PUSHes y POPs de registros de 16 bits. Sea el siguiente programa:

```
LD BC, 00FFh
LD DE, AABBh
LD SP, 65535     ; Puntero de pila al final de la memoria
```

Si ahora hacemos:

```
PUSH BC          ; Apilamos el registro BC
```

Lo que estaremos haciendo es:

```
SP = SP - 2 = 65533
(SP) = BC = 00FFh
```

Con lo que el contenido de la memoria sería:

| | **Celdilla** | **Contenido** |
|---|---|---|
| | 65534 | FFh |
| SP -> | 65533 | 00h |

Si a continuación hacemos otro PUSH:

```
PUSH DE          ; Apilamos el registro DE
```

Lo que estaremos haciendo es:

```
SP = SP - 2 = 65531
(SP) = DE = AABBh
```

Con lo que el contenido de las celdillas de memoria sería:

| | **Celdilla** | **Contenido** |
|---|---|---|
| | 65534 | FFh |
| | 65533 | 00h |

| | | |
|---|---|---|
| | 65532 | AAh |
| SP -> | 65531 | BBh |

Si ahora hacemos un POP:

```
POP DE
```

Lo que hacemos es:

```
DE = (SP) = AABBh
SP = SP + 2 = 65533
```

Y la memoria queda, de nuevo, como:

| | **Celdilla** | **Contenido** |
|---|---|---|
| | 65534 | FFh |
| SP -> | 65533 | 00h |

Como podemos ver, *PUSH va apilando valores*, haciendo decrecer el valor de SP, mientras que *POP recupera valores*, haciendo crecer (en 2 bytes, 16 bits) el valor de SP.

**PUSH y POP**

Así pues, podemos hacer **PUSH** y **POP** de los siguientes registros:

- PUSH: AF, BC, DE, HL, IX, IY
- POP : AF, BC, DE, HL, IX, IY

Lo que hacen PUSH y POP, tal y como funciona la pila, es:

```
PUSH xx :
  SP   = SP-2
  (SP) = xx

POP xx :
  xx   = (SP)
  SP   = SP+2
```

Nótese cómo la pila se decrementa ANTES de poner los datos en ella, y se incrementa DESPUES de sacar datos de la misma. Esto mantiene siempre SP apuntando al TOS (*Top Of Stack*).

```
                        Flags
  Instrucción       |S Z H P N C|
----------------------------------
POP xx              |- - - - - -|
PUSH xx             |- - - - - -|
```

Nótese que también podemos apilar y desapilar AF. De hecho, es una forma de manipular los bits del registro F (hacer PUSH BC con un valor determinado, por ejemplo, y hacer un POP AF).

**Utilidad de la pila del Spectrum**

La pila resulta muy útil para gran cantidad de tareas en programas en ensamblador. Veamos algunos ejemplos:

- Intercambiar valores de registros mediante PUSH y POP. Por ejemplo, para intercambiar el valor de BC y de DE:

```
PUSH BC  ; Apilamos BC
PUSH DE  ; Apilamos DE
POP BC   ; Desapilamos BC
         ; ahora BC=(valor apilado de DE)
```

```
 POP DE   ; Desapilamos DE
          ; ahora DE=(valor apilado de BC)
```

- POP AF es la principal forma de manipular el registro F directamente (haciendo PUSH de otro registro y POP de AF).
- Almacenaje de datos mientras ejecutamos porciones de código: Supongamos que tenemos un registro cuyo valor queremos mantener, pero que tenemos que ejecutar una porción de código que lo modifica. Gracias a la pila podemos hacer lo siguiente:

```
 PUSH BC    ; Guardamos el valor de BC
 (código)   ; Hacemos operaciones
 POP BC     ; Recuperamos el valor que teníamos en BC
```

Esto incluye, por ejemplo, el almacenaje del valor de BC en los bucles cuando necesitamos operador con B, C o BC:

```
    LD A, 0
    LD B, 100
bucle:
    PUSH BC     ; Guardamos BC
    LD B, 1
    ADD A, B
    POP BC      ; Recuperamos BC
    DJNZ bucle
```

En este sentido, también podremos anidar 2 o más bucles que usen el registro B o BC con PUSH y POPs entre ellos. Supongamos un bucle BASIC del tipo:

```
  FOR I=0 TO 20:
    FOR J=0 TO 100:
       CODIGO
    NEXT J
  NEXT I
```

En ensamblador podríamos hacer:

```
    LD B, 20                ; repetimos bucle externo 20 veces

bucle_externo:
    PUSH BC                 ; Nos guardamos el valor de BC
    LD B, 100               ; Iteraciones del bucle interno
bucle_interno:
    (... código ...)
    DJNZ bucle_interno      ; FOR J=0 TO 100
    POP BC                  ; Recuperamos el valor de B

    DJNZ bucle_externo      ; FOR I=0 TO 20
```

Hay que tener en cuenta que PUSH y POP implican escribir en memoria (en la dirección apuntada por SP), por que siempre serán más lentas que guardarse el valor actual de B en otro registro:

```
    LD B, 20                ; repetimos bucle externo 20 veces

bucle_externo:
    LD D, B                 ; Nos guardamos el valor de B

    LD B, 100               ; Iteraciones del bucle interno
bucle_interno:
    (... código ...)        ; En este codigo no podemos usar D
    DJNZ bucle_interno      ; FOR J=0 TO 100
```

```
    LD B, D                  ; Recuperamos el valor de B

    DJNZ bucle_externo       ; FOR I=0 TO 20
```

No obstante, en múltiples casos nos quedaremos sin registros libres donde guardar datos, por lo que la pila es una gran opción. No hay que obsesionarse con no usar la pila porque implique escribir en memoria. A menos que estemos hablando de una rutina muy muy crítica, que se ejecute muchas veces por cada fotograma de nuestro juego, PUSH y POP serán las mejores opciones para preservar valores.

- Almacenaje de datos de entrada y salida en subrutinas: Podemos pasar parámetros a nuestras rutinas apilándolos en el stack, de forma que nada más entrar en la rutina leamos de la pila esos parámetros.
- Extendiendo un poco más el punto anterior, cuando realicemos funciones en ensamblador embebidas dentro de otros lenguajes (por ejemplo, dentro de programas en C con Z88DK), podremos recoger dentro de nuestro bloque en ensamblador los parámetros pasados con llamadas de funciones C.
- Como veremos en el próximo apartado, la pila es la clave de las subrutinas (CALL/RET) en el Spectrum (equivalente al GOSUB/RETURN de BASIC).

Recordad también que tenéis instrucciones de intercambio (EX) que permiten manipular la pila. Hablamos de:

```
EX (SP), HL
EX (SP), IX
EX (SP), IY
```

**Los peligros de la pila**

Pero como todo arma, las pilas también tienen un doble filo. Mal utilizada puede dar lugar a enormes desastres en nuestros programas. Veamos algunos de los más habituales:

- Dado que la pila decrece en memoria, tenemos que tener cuidado con el valor de SP y la posición más alta de memoria donde hayamos almacenado datos o rutinas. Si ponemos un gráfico o una rutina cerca del valor inicial de SP, y realizamos muchas operaciones de PUSH, podemos machacar nuestros datos con los valores de los registros que estamos apilando.
- Hacer más PUSH que POP o más POP que PUSH. Recordemos que la pila tiene que ser consistente. Si hacemos un push, debemos recordar hacer el pop correspondiente (a menos que haya una razón para ello), y viceversa. Como veremos a continuación, la pila es utilizada tanto para pasar parámetros a funciones como para volver de ellas, si introducimos un valor en ella con PUSH dentro de una función y no lo sacamos antes de hacer el RET, nuestro programa continuará su ejecución el algún lugar de la memoria que no era al que debía volver. Es más, si nuestro programa debe volver a BASIC correctamente tras su ejecución, entonces es obligatorio que hagamos tantos PUSH como POP para que el punto final de retorno del programa al BASIC esté en la siguiente posición de la pila cuando nuestro programa acabe.
- Ampliando la regla anterior, hay que tener cuidado con los bucles a la hora de hacer PUSH y POP.
- Finalmente, no hay que asumir que SP tiene un valor correcto para nosotros. Tal vez tenemos planeado usar una zona de la memoria para guardar datos o subrutinas y el uso de PUSH y POP pueda machacarlo. Si sabemos dónde no puede hacer daño SP y sus escrituras en memoria, basta con inicializar la pila al principio de nuestro programa a una zona de memoria libre (por ejemplo, "LD SP, 23999", o cualquier otra dirección que sepamos que no vamos a usar). Esto no es obligatorio y muchas veces el valor por defecto de SP será válido, siempre que no usemos zonas de la memoria que creemos libres como "almacenes temporales". Si usamos "variables" creadas en tiempo de ensamblado (definidas como DB o DW en el ensamblador) no deberíamos tener problemas, al menos con programas pequeños.

Veamos algunos ejemplos de "errores" con la pila. Empecemos con el típico PUSH del cual se nos olvida hacer POP:

```
; Este programa se colgará (probablemente, depende de BC)
; pero en cualquier caso, no seguirá su ejecución normal.
PUSH BC
PUSH DE

(código)

POP DE
RET          ; En lugar de volver a la dirección de memoria
             ; a la que teníamos que volver, volveremos a
             ; la dirección apuntada por el valor de BC, que
             ; no hemos recogido de la pila.
```

También hay que tener cuidado con los bucles:

```
bucle:
    PUSH BC          ; Nos queremos guardar BC
    (código que usa B)

    JR flag, bucle
    POP BC
```

En ese código hacemos múltiples PUSHes pero un sólo POP. Probablemente, en realidad, queremos hacer lo siguiente:

```
    PUSH BC          ; Nos queremos guardar BC
bucle:
    (código)

    JR flag, bucle
    POP BC
```

Y una curiosidad al respecto de la pila y la sentencia **CLEAR** de BASIC: en el fondo, lo que realiza esta función es cambiar el valor de la variable del sistema RAMTOP, lo que implica cambiar el valor de SP. Así, con **CLEAR XXXX**, *ponemos la pila colgando de la dirección de memoria XXXX*, asegurándonos de que BASIC no pueda hacer crecer la pila de forma que machaque código máquina que hayamos cargado nosotros en memoria. Si, por ejemplo, vamos a cargar todo nuestro código a partir de 50000, en nuestro cargador BASIC haremos un CLEAR 49999, de forma que BASIC no podrá tocar ninguna dirección de memoria por encima de este valor.

**Rutinas: CALL y RET**

Ya de por sí el lenguaje ensamblador es un lenguaje de listados "largos" y enrevesados, y donde teníamos 10 líneas en BASIC podemos tener 100 ó 1000 en ensamblador.

Lo normal para hacer el programa más legible es *utilizar bloques de código que hagan unas funciones concretas* y a los cuales podamos llamar a lo largo de nuestro programa. Esos bloques de código son las **funciones o subrutinas**.

Las subrutinas son bloques de código máquina a las cuales saltamos, hacen su tarea asignada, y devuelven el control al punto en que fueron llamadas. A veces, esperan recibir los registros con una serie de valores y devuelven registros con los valores resultantes.

Por ejemplo:

```
; SUMA_A_10
;
; SUMA 10 a A y devuelve el resultado en B
;
; Nota: Modifica el valor de A

SUMA_A_10:
   ADD A, 10         ; A = A + 10
   LD B, A           ; B = A
```

Nuestra función/subrutina de ejemplo espera obtener en A un valor, y devuelve el resultado de su ejecución en B. Antes de llamar a esta rutina, nosotros deberemos poner en A el valor sobre el que actuar, y posteriormente interpretar el resultado (sabiendo que lo tenemos en B).

Pero, ¿cómo llamamos a las subrutinas y volvemos de ellas? Podéis pensar en nuestro ejemplo anterior y la orden "**JP**":

```
    LD A, 35
    JP SUMA_A_10
volver1:

   (...)
SUMA_A_10:
   ADD A, 10         ; A = A + 10
   LD B, A           ; B = A
   JP volver1        ; Volvemos de la subrutina
```

En este caso, cargaríamos A con el valor 35, saltaríamos a la subrutina, sumaríamos 10 a A (pasando a valer 45), haríamos B = 45, y volveríamos al lugar posterior al punto de llamada. Pero … ¿qué pasaría si quisieramos volver a llamar a la subrutina desde otro punto de nuestro programa? Que sería inviable, porque nuestra subrutina acaba con un "JP volver1" que no devolvería la ejecución al punto desde donde la hemos llamado, sino a "volver1".

```
    LD A, 35
    JP SUMA_A_10
volver1:

    LD A, 50
    JP SUMA_A_10
                     ; Nunca llegariamos a volver aqui
   (...)
SUMA_A_10:
   ADD A, 10         ; A = A + 10
   LD B, A           ; B = A
   JP volver1        ; Volvemos de la subrutina
```

Para evitar ese enorme problema es para lo que se usa **CALL** y **RET**.

**Uso de CALL y RET**

**CALL** es, en esencia, similar a JP, salvo porque antes de realizar el salto, introduce en la pila (PUSH) el valor del registro PC (Program Counter, o contador de programa), el cual una vez leído y decodificado el CALL apunta a la siguiente instrucción tras el mismo.

¿Y para qué sirve eso? Para que lo aprovechemos dentro de nuestra subrutina con **RET**. RET lee de la pila la dirección que introdujo CALL y salta a ella. Así, cuando acaba nuestra función, el RET devuelve la ejecución a la instrucción siguiente al CALL que hizo la llamada.

Son, por tanto, el equivalente ensamblador de GO SUB y RETURN en BASIC (o más bien se debería decir que GO SUB y RETURN son la implantación en BASIC de estas instrucciones del microprocesador).

```
 CALL NN equivale a:
    PUSH PC
    JP NN

 RET equivale a:
    POP PC
```

```
Veamos la aplicación de CALL y RET con nuestro ejemplo anterior:
```

```
    LD A, 35
    CALL SUMA_A_10

    LD A, 50
    CALL SUMA_A_10

    LD C, B

   (...)

SUMA_A_10:
   ADD A, 10         ; A = A + 10
   LD B, A           ; B = A
   RET               ; Volvemos de la subrutina
```

En esta ocasión, cuando ejecutamos el primer CALL, se introduce en la pila el valor de PC, que se corresponde exáctamente con la dirección de memoria donde estaría ensamblada la siguiente instrucción (LD A, 50). El CALL cambia el valor de PC al de la dirección de "SUMA_A_10", y se continúa la ejecución dentro de la subrutina.

Al acabar la subrutina encontramos el RET, quien extrae de la pila el valor de PC anteriormente introducido, con

lo que en el siguiente ciclo de instrucción del microprocesador, el Z80 leerá, decodificará y ejecutará la instrucción "LD A, 50", siguiendo el flujo del programa linealmente desde ahí. Con la segunda llamada a CALL ocurriría lo mismo, pero esta vez lo que se introduce en la pila es la dirección de memoria en la que está ensamblada la instrucción "LD C, B". Esto asegura el retorno de nuestra subrutina al punto adecuado.

Al hablar de la pila os contamos lo importante que era mantener la misma cantidad de PUSH que de POPs en nuestro código. Ahora entenderéis por qué: si dentro de una subrutina hacéis un PUSH que no elimináis después con un POP, cuando lleguéis al RET éste obtendrá de la pila un valor que no será el introducido por CALL, y saltará allí. Por ejemplo:

```
  CALL SUMA_A_10
  LD C, B              ; Esta dirección se introduce en la pila con CALL

SUMA_A_10:
  LD DE, $0000
  PUSH DE
  ADD A, 10
  LD B, a
  RET                  ; RET no sacará de la pila lo introducido por CALL
                       ; sino "0000", el valor que hemos pulsado nosotros.
```

Aquí RET sacará de la pila 0000h, en lugar de la dirección que introdujo CALL, y saltará al inicio del a ROM, produciendo un bonito reset.

Ni call ni RET afectan a la tabla de flags del registro F.

```
                        Flags
   Instrucción       |S Z H P N C|
 ----------------------------------
 CALL NN             |- - - - - -|
 RET                 |- - - - - -|
```

Saltos y retornos condicionales

Una de las peculiaridades de CALL y RET es que tienen instrucciones condicionales con respecto al estado de los flags, igual que "JP cc" o "JR cc", de forma que podemos condicionar el SALTO (CALL) o el retorno (RET) al estado de un determinado flag.

Para eso, utilizamos las siguientes instrucciones:

- **CALL flag, NN** : Salta sólo si FLAG está activo.
- **RET flag** : Vuelve sólo si FLAG está activo.

Por ejemplo, supongamos que una de nuestras subrutinas tiene que comprobar que uno de los parámetros que le pasamos, BC, no sea 0.

```
; Copia_Pantalla:
;
; Entrada:
;             HL = direccion origen
;             DE = direccion destino
;             BC = bytes a copiar
;
Copia_Pantalla:

   ; lo primero, comprobamos que BC no sea cero:
   LD A, B
   OR C                           ; Hacemos un OR de B sobre C
                                  ; Si BC es cero, activará el flag Z
   RET Z                          ; Si BC es cero, volvemos sin hacer nada

   (más código)
   ; Aquí seguiremos si  BC no es  cero, el
   ; RET no se habrá ejecutado.
```

Del mismo modo, el uso de CALL condicionado al estado de flags (CALL Z, CALL NZ, CALL M, CALL P, etc) nos permitirá llamar o no a funciones según el estado de un flag.

Al igual que CALL y RET, sus versiones condicionales no afectan al estado de los flags.

```
                         Flags
   Instrucción       |S Z H P N C|         Pseudocodigo
 -----------------------------------------------------------
 CALL cc, NN         |- - - - - -|        IF cc CALL NN
 RET cc              |- - - - - -|        IF cc RET
```

**Pasando parametros a rutinas**

Ahora que ya sabemos crear rutinas y utilizarlas, vamos a ver los 3 métodos que hay para pasar y devolver parámetros a las funciones.

*Método 1: Uso de registros*

Este método consiste en modificar unos registros concretos antes de hacer el CALL a nuestra subrutina, sabiendo que dicha subrutina espera esos registros con los valores sobre los que actuar. Asímismo, nuestra rutina puede modificar alguno de los registros con el objetivo de devolvernos un valor.

Por ejemplo:

```
; Función de multiplicación de 8 bits
;
; Entrada:
;           H = valor 8 bits
;           E = valor 8 bits
; Salida:
;           HL = valor 16 bits H*E
; Modifica:
;           B, D, L, FLAGS
;
MULTIPLICA:
   LD L, 0
   LD D, L    ; L=0 AND D=0
   LD B, 8
MULT_LOOP:
   ADD HL, HL
   JR NC, NOADD
   ADD HL, DE
NOADD:
   DJNZ MULT_LOOP
```

Antes de hacer la llamada a MULTIPLICA, tendremos que cargar en H y en E los valores que queremos multiplicar, de modo que si estos valores están en otros registros o en memoria, tendremos que moverlos a H y a E.

Además, sabemos que la salida nos será devuelta en HL, con lo que si dicho registro (especialmente L en nuestro caso, ya que H es un parámetro de entrada) contiene algún valor importante, deberemos preservarlo previamente.

Con este tipo de funciones resulta importantísimo realizarse cabeceras de comentarios explicativos, que indiquen:

- a.- Qué función realiza la subrutina.
- b.- Qué registros espera como entrada.
- c.- Qué registros devuelve como salida.
- d.- Qué registros modifica además de los de entrada y salida.

Con este tipo de paso de parámetros tenemos el mayor ahorro y la mayor velocidad: no se accede a la pila y no se accede a la memoria, pero por contra tenemos que tenerlo todo controlado. Tendremos que saber en cada momento qué parámetros de entrada y de salida utiliza (de ahí la importancia del comentario explicativo, al que acudiremos más de una vez cuando no recordemos en qué registros teníamos que pasarle los datos de entrada), y asegurarnos de que ninguno de los registros "extra" que modifica están en uso antes de llamar a la función, puesto que se verán alterados.

Si no queremos que la función modifique muchos registros además de los de entrada y salida, siempre podemos poner una serie de PUSH y POP en su inicio y final, al estilo:

```
MiFuncion:
```

```
    PUSH BC
    PUSH DE      ; Nos guardamos sus valores

    (...)

    POP DE
    POP BC       ; Recuperamos sus valores
    RET
```

En funciones que no sean críticas en velocidad, es una buena opción porque no tendremos que preocuparnos por el estado de nuestros registros durante la ejecución de la subrutina: al volver de ella tendrán sus valores originales (excepto aquellos de entrada y salida que consideremos necesarios).

No nos olvidemos de que en algunos casos podemos usar el juego de registros alternativos (EX AF, AF', EXX) para evitar algún PUSH o POP.

*Método 2: Uso de localidades de memoria*

Aunque no es una opción rápida, sí que es bastante efectivo y sencillo el uso de variables o posiciones de memoria para pasar y recoger parámetros de funciones. Nos ahorra el uso de muchos registros, y hace que podamos usar dentro de las funciones prácticamente todos los registros. Se hace especialmente útil usando el juego de registros alternativos.

Por ejemplo:

```
      LD A, 10
      LD (x), A
      LD A, 20
      LD (y), A
      LD BC, 40
      LD (size), BC      ; Parametros de entrada a la funcion
      CALL MiFuncion
      (...)

MiFuncion:
      EXX                ; Preservamos TODOS los registros

      LD A, (x)
      LD B, A
      LD A, (y)
      LD BC, (size)      ; Leemos los parametros

      (Codigo)

      LD (salida), a     ; Devolvemos un valor
      EXX

x      DB  0
y      DB  0
size   DW  0
salida DB  0
```

Este es un ejemplo exagerado donde todos los parámetros se pasan en variables, pero lo normal es usar un método mixto entre este y el anterior, pasando cosas en registros excepto si nos quedamos sin ellos (por que una función requiere muchos parámetros, por ejemplo), de forma que algunas cosas las pasamos con variables de memoria.

*Método 3: Uso de la pila (método C)*

El tercer método es el sistema que utilizan los lenguajes de alto nivel para pasar parámetros a las funciones: el apilamiento de los mismos. Este sistema no se suele utilizar en ensamblador, pero vamos a comentarlo de forma que os permita integrar funciones en ASM dentro de programas escritos en C, como los compilables con el ensamblador Z88DK.

En C (y en otros lenguajes de programación) los parámetros se insertan en la pila en el orden en que son leídos. La subrutina debe utilizar el registro SP (una copia) para acceder a los valores apilados en orden inverso. Estos valores son siempre de 16 bits aunque las variables pasadas sean de 8 bits (en este caso ignoraremos el byte que no contiene datos, el segundo).

Veamos unos ejemplos:

```
//-----------------------------------------------------------------
// Sea parte de nuestro programa en C:

  int jugador_x, jugador_y;

  jugador_x = 10;
  jugador_y = 200;
  Funcion( jugador_x, jugador_y );
  (...)



//-----------------------------------------------------------------
int Funcion( int x, int y )
{

#asm
   LD HL,2
   ADD HL,SP            ; Ahora SP apunta al ultimo parametro metido
                        ; en la pila por el compilador (valor de Y)

   LD C, (HL)
   INC HL
   LD B, (HL)
   INC HL               ; Ahora BC = y

   LD E, (HL)
   INC HL
   LD D, (HL)
   INC HL               ; Ahora, DE = x

   (ahora hacemos lo que queramos en asm)

#endasm
}
```

No tenemos que preocuparnos por hacer PUSH y POP de los registros para preservar su valor dado que C lo hace automáticamente antes y después de cada #asm y #endasm.

El problema es que conforme crece el número de parámetros apilados, es posible que tengamos que hacer malabarismos para almacenarlos, dado que no podemos usar HL (es nuestro puntero a la pila en las lecturas). Veamos el siguiente ejemplo con 3 parámetros, donde tenemos que usar PUSH para guardar el valor de DE y EX DE, HL para acabar asociando el valor final a HL:

```
//-----------------------------------------------------------------
int Funcion( int x, int y, int z )
{

#asm
   LD HL,2
   ADD HL,SP            ; Ahora SP apunta al ultimo parametro metido
                        ; en la pila por el compilador (tamanyo)

   LD C, (HL)
   INC HL
   LD B, (HL)
   INC HL               ; Ahora BC = z

   LD E, (HL)
   INC HL
   LD D, (HL)
   INC HL               ; Ahora, DE = y

   PUSH DE              ; Guardamos DE

   LD E, (HL)
   INC HL
```

```
   LD D, (HL)
   INC HL              ; Usamos DE para leer el valor de x

   EX DE, HL           ; Ahora cambiamos x a HL
   POP DE              ; Y recuperamos el valor de y en DE

   ; (ahora hacemos lo que queramos en asm)

#endasm
}
```

La manera de leer bytes (char) pulsados en C es de la misma forma que leemos una palabra de 16 bits, pero ignorando la parte alta. En realidad, como la pila es de 16 bits, el compilador convierte el dato de 8 bits en uno de 16 (rellenando con ceros) y pulsa este valor:

```
//-----------------------------------------------------------------
int Funcion( char x, y )
{

#asm
   LD HL,2
   ADD HL,SP           ; Ahora SP apunta al ultimo parametro metido
                       ; en la pila por el compilador (tamanyo)

   LD A, (HL)          ; Aquí tenemos nuestro dato de 8 bits (y)
   LD B, A
   INC HL
   INC HL              ; La parte alta del byte no nos interesa

   LD A, (HL)          ; Aquí tenemos nuestro dato de 8 bits (x)
   LD C, A
   INC HL
   INC HL              ; La parte alta del byte no nos interesa

   (ahora hacemos lo que queramos en asm)

#endasm
}
```

En ocasiones, es posible que incluso tengamos que utilizar variables auxiliares de memoria para guardar datos:

```
//-----------------------------------------------------------------
int Funcion( int x, int y, char z )
{

#asm
   LD HL,2
   ADD HL,SP           ; Ahora SP apunta al ultimo parametro metido
                       ; en la pila por el compilador (tamanyo)

   LD C, (HL)
   INC HL
   LD B, (HL)
   INC HL              ; Ahora BC = y
   LD (valor_y), BC    ; nos lo guardamos, BC libre de nuevo

   LD C, (HL)
   INC HL
   LD B, (HL)
   INC HL
   LD (valor_x), BC    ; Nos lo guardamos, BC libre de nuevo

   LD A, (HL)
   LD (valor_z), A     ; Nos guardamos el byte
   INC HL
```

```
   INC HL              ; La parte alta del byte no nos interesa

   (ahora hacemos lo que queramos en asm)

   RET

valor_x  defw  0
valor_y  defw  0
valor_z  defb  0

#endasm
}
```

Por contra, para devolver valores no se utiliza la pila (dado que no podemos tocarla), sino que se utiliza un determinado registro. En el caso de Z88DK, se utiliza el registro HL. Si la función es de tipo INT o CHAR en cuanto a devolución, el valor que dejemos en HL será el que se asignará en una llamada de este tipo:

```
 valor = MiFuncion_ASM( x, y, z);
```

Hemos considerado importante explicar este tipo de paso de parámetros y devolución de valores porque nos permite integrar nuestro código ASM en programas en C.

**Integracion de ASM en Z88DK**

Para aprovechar esta introducción de "uso de ASM en z88dk", veamos el código de alguna función en C que use ASM internamente y que muestre, entre otras cosas, la lectura de parámetros de la pila, el acceso a variables del código C, el uso de etiquetas, o la devolución de valores.

```
//
// Devuelve la direccion de memoria del atributo de un caracter
// de pantalla, de coordenadas (x,y). Usando la dirección que
// devuelve esta función (en HL, devuelto en la llamada), podemos
// leer o cambiar los atributos de dicho carácter.
//
// Llamada:   valor =  Get_LOWRES_Attrib_Address( 1, 3 );
//
int Get_LOWRES_Attrib_Address( char x, char y )
{
#asm

   LD HL, 2
   ADD HL, SP                  ; Leemos x e y de la pila
   LD  D, (HL)  ; d = y
   INC HL                      ; Primero "y" y luego "x".
   INC HL                      ; Como son "char", ignoramos parte alta.
   LD  E, (HL)  ; e = x

   LD H, 0
   LD L, D
   ADD HL, HL                  ; HL = HL*2
   ADD HL, HL                  ; HL = HL*4
   ADD HL, HL                  ; HL = HL*8
   ADD HL, HL                  ; HL = HL*16
   ADD HL, HL                  ; HL = HL*32
   LD D, 0
   ADD HL, DE                  ; Ahora HL = (32*y)+x
   LD BC, 16384+6144           ; Ahora BC = offset attrib (0,0)
   ADD HL, BC                  ; Sumamos y devolvemos en HL

#endasm
}

//
// Set Border
// Ejemplo de modificación del borde, muestra cómo leer variables
// globales de C en ASM, añadiendo "_" delante.
//
```

```
char my_tmp_border;

void BORDER( unsigned char value )
{
   my_tmp_border = value<<3;
#asm
   LD HL, 2
   ADD HL, SP
   LD A, (HL)
   LD C, 254
   OUT (C), A
   LD HL, 23624
   LD A, (_my_tmp_border)
   LD (HL), A
#endasm
}


//
// Realización de un fundido de la pantalla hacia negro
// Con esta función se muestra el uso de etiquetas. Nótese
// como en lugar de escribirse como ":", se escriben sin
// ellos y con un punto "." delante.
//
void FadeScreen( void )
{

#asm

   LD B, 16

.fadescreen_loop1
   LD HL, 16384+6144
   LD DE, 768

.fadescreen_loop2
   LD A, (HL)                  ; get attr

   LD C, A
   AND 7                       ; get last  bits
   JR Z, fadescreen_ink_zero   ; if its zero, not dec it

   DEC A
   EX AF, AF

.fadescreen_ink_zero
   LD A, C
   AND 7
   SRA A
   SRA A
   SRA A
   JR Z, fadescreen_paper_zero   ; if its zero, not dec it

   DEC A

.fadescreen_paper_zero
   SLA A
   SLA A
   SLA A
   LD C, A
   EX AF, AF
   ADD A, C

   LD (HL), A
   INC HL

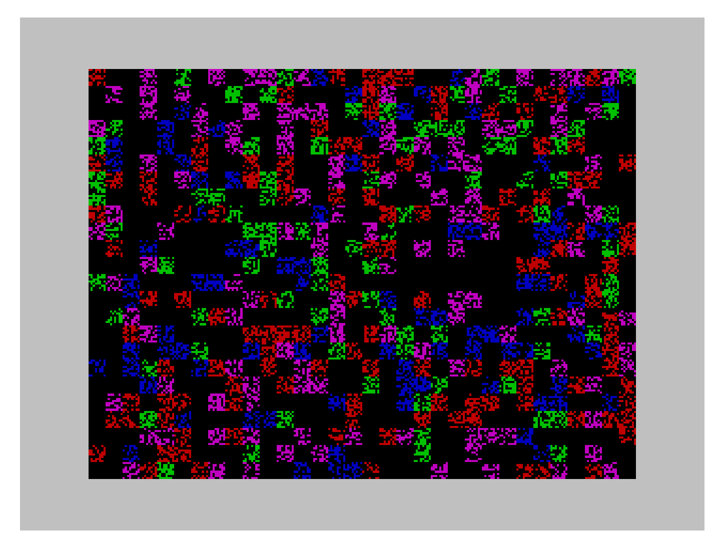
   DEC DE                       ; internal loop
   LD A, D
   OR E
```

```
    JP NZ, fadescreen_loop2

    DJNZ fadescreen_loop1

#endasm
}
```

Si tenéis curiosidad por ver el funcionamiento de esta rutina de Fade (fundido), podéis verla integramente en ASM en el fichero fade.asm. Un detalle a tener en cuenta, en Z88DK se soporta "EX AF, AF", mientras que pasmo requiere poner la comilla del shadow-register: "EX AF, AF'".



*Captura durante el fade de la pantalla*

En la anterior captura podéis ver el aspecto de uno de los pasos del fundido.

**La importancia de usar subrutinas**

Usar subrutinas es mucho más importante de lo que parece a simple vista: *nos permite organizar el programa en unidades o módulos funcionales* que cumplen una serie de funciones específicas, lo que hace mucha más sencilla su depuración y optimización.

Si en el menú de nuestro juego estamos dibujando una serie de sprites móviles, y también lo hacemos a lo largo del juego, resulta absurdo "construir" 2 bloques de código, uno para mover los sprites del menú y otro para los del juego. Haciendo esto, si encontramos un error en una de las 2 rutinas, o realizamos una mejora, deberemos corregirlo en ambas.

Por contra, si creamos una subrutina, digamos, DrawSprite, que podamos llamar con los parámetros adecuados en ambos puntos del programa, cualquier cambio, mejora o corrección que realicemos en DrawSprite afectará a todas las llamadas que le hagamos. También reducimos así el tamaño de nuestro programa (y con él el tiempo de carga del mismo), las posibilidades de fallo, y la longitud del listado (haciéndolo más legible y manejable).

Aunque no sea el objetivo de esta serie de artículos, antes de sentarse a teclear, un buen programador debería coger un par de folios de papel y hacer un pequeño análisis de lo que pretende crear. Este proceso, la fase de diseño, define qué debe de hacer el programa y, sobre todo, una división lógica de cuáles son las principales partes del mismo. Un sencillo esquema en papel, un diagrama de flujo, identificar las diferentes partes del programa, etc.

El proceso empieza con un esbozo muy general del programa, que será coincidente con la gran mayoría de los juegos: inicialización de variables, menú (que te puede llevar bien a las opciones o bien al juego en sí), y dentro del juego, lectura de teclado/joystick, trazado de la pantalla, lógica del juego, etc.

Después, se teclea un programa vacío que siga esos pasos, pero que no haga nada; un bucle principal que tenga un aspecto parecido a:

```
BuclePrincipal:
     CALL Leer_Teclado
     CALL Logica_Juego
     CALL Comprobar_Estado
     jp Bucle_Principal

Leer_Teclado:
     RET

Logica_Juego:
```

```
    RET

Comprobar_Estado:
    RET
```

Tras esto, ya tenemos el "esqueleto del programa". Y ahora hay que rellenar ese esqueleto, y la mejor forma de hacerlo es aprovechar esa "modularidad" que hemos obtenido con ese diseño en papel.

Por ejemplo, supongamos que nuestro juego tiene que poder dibujar sprites y pantallas hechas a bases de bloques que se repiten (tiles). Gracias a nuestro diseño, sabemos que necesitamos una rutina que imprima un sprite, una rutina que dibuje un tile y una rutina que dibuje una pantalla llena de tiles.

Pues bien, creamos un programa en ASM nuevo, desde cero, y en él creamos una función DrawSprite que acepte como parámetros la dirección origen de los datos del Sprite, y las posiciones X e Y donde dibujarlo, y la realizamos. En este nuevo programa, pequeño, sencillo de leer, realizamos todo tipo de pruebas:

```
   ORG 50000

   ; Probamos de diferentes formas nuestra rutina
   LD B, 10
   LD C, 15
   LD HL, sprite
   CALL DrawSprite
   RET

; Rutina DrawSprite
; Acepta como parametros ... y devuelve ...
DrawSprite:
   (aquí el código)
   RET

sprite DB 0,0,255,123,121,123,34, (etc...)
```

Gracias a esto, podremos probar nuestra nueva rutina y trabajar con ella limpiamente y en un fichero de programa pequeño. Cuando la tenemos lista, basta con copiarla a nuestro programa "principal" y ya sabemos que la tenemos disponible para su uso con CALL.

Así, vamos creando diferentes rutinas en un entorno controlado y testeable, y las vamos incorporando a nuestro programa. Si hay algún bug en una rutina y tenemos que reproducirlo, podemos hacerlo en nuestros pequeños programas de prueba, evitando el típico problema de tener que llegar a un determinado punto de nuestro programa para chequear una rutina, o modificar su bucle principal para hacerlo.

Además, el definir de antemano qué tipo de subrutinas necesitamos y qué parámetros deben aceptar o devolver permite trabajar en equipo. Si sabes que necesitarás una rutina que dibuje un sprite, o que lea el teclado y devuelva la tecla pulsada, puedes decir los registros de entrada y los valores de salida que necesitas, y que la realice una segunda persona y te envíe la rutina lista para usar.

En ocasiones una excesiva desgranación del programa en módulos más pequeños puede dar lugar a una penalización en el rendimiento, aunque no siempre es así. Por ejemplo, supongamos que tenemos que dibujar un mapeado de 10×10 bloques de 8×8 pixeles cada uno. Si hacemos una función de que dibuja un bloque de 8×8, podemos llamarla en un bucle para dibujar nuestros 10×10 bloques.

Hay gente que, en lugar de esto, preferirá realizar una función específica que dibuje los 10×10 bloques dentro de una misma función. Esto es así porque de este modo te evitas 100 CALLs (10×10) y sus correspondientes RETs, lo cual puede ser importante en una rutina gráfica que se ejecute X veces por segundo. Por supuesto, en muchos casos tendrán razón, en ciertas ocasiones hay que hacer rutinas concretas para tareas concretas, aún cuando puedan repetir parte de otro código que hayamos escrito anteriormente, con el objetivo de evitar llamadas, des/apilamientos u operaciones innecesarias en una función crítica.

Pero si, por ejemplo, nosotros sólo dibujamos la pantalla una vez cuando nuestro personaje sale por el borde, y no volvemos a dibujar otra hasta que sale por otro borde (típico caso de juegos sin scroll que muestran pantallas completas de una sóla vez), vale la pena el usar funciones modulares dado que unos milisegundos más de ejecución en el trazado de la pantalla no afectarán al desarrollo del juego.

Al final hay que llegar a un compromiso entre modularidad y optimización, en algunos casos nos interesará desgranar mucho el código, y en otros nos interesará hacer funciones específicas. Y esa decisión no deja de ser, al fin y al cabo, diseño del programa.

En cualquier caso, el diseño nos asegura que podremos implementar nuestro programa en cualquier lenguaje y en cualquier momento. Podremos retomar nuestros "papeles de diseño" 3 meses después y, pese a no recordar en qué parte del programa estábamos, volver a su desarrollo sin excesivas dificultades.

Una de las cosas más complicadas de hacer un juego es el pensar por dónde empezar. Todo este proceso nos permite empezar el programa por la parte del mismo que realmente importa. Todos hemos empezado alguna vez a realizar nuestro juego por el menú, perdiendo muchas horas de trabajo para descubrir que teníamos un menú, pero no teníamos un juego, y que ya estábamos cansados del desarrollo sin apenas haber empezado.

Veamos un ejemplo: suponiendo que realizamos, por ejemplo, un juego de puzzles tipo Tetris, lo ideal sería empezar definiendo dónde se almacenan los datos del area de juego, hacer una función que convierta esos datos en imágenes en pantalla, y realizar un bucle que permita ver caer la pieza. Después, se agrega control por teclado para la pieza y se pone la lógica del juego (realización de líneas al tocar suelo, etc).

Tras esto, ya tenemos el esqueleto funcional del juego y podemos añadir opciones, menúes y demás. Tendremos algo tangible, funcional, donde podemos hacer cambios que implican un inmediato resultado en pantalla, y no habremos malgastado muchas horas con un simple menú.

Por otra parte, el diseñar correctamente nuestro programa y desgranarlo en piezas reutilizables redundará en nuestro beneficio no sólo actual (con respecto al programa que estamos escribiendo) sino futuro, ya que podremos crearnos nuestras propias "bibliotecas" de funciones que reutilizar en futuros programas. Aquella rutina de dibujado de Sprites, de zoom de pantalla o de compresión de datos que tanto nos costó programar, bien aislada en una subrutina y con sus parámetros de entrada y salida bien definidos puede ser utilizada directamente en nuestros próximos programas simplemente copiando y pegando el código correspondiente.

Más aún, podemos organizar funciones con finalidades comunes en ficheros individuales. Tendremos así nuestro fichero / biblioteca con funciones gráficas, de sonido, de teclado/joystick, etc. El ensamblador PASMO nos permite incluir un fichero en cualquier parte de nuestro código con la directiva "**INCLUDE**".

Así, nuestro programa en ASM podría comenzar (o acabar) por algo como:

```
INCLUDE "graficos.asm"
INCLUDE "sonido.asm"
INCLUDE "teclado.asm"
INCLUDE "datos.asm"
```

Esto contribuye a reducir fallos en la codificación, hacer más corto el "listado general del programa", y, sobre todo, reduce el tiempo de desarrollo.

# Lenguaje Ensamblador del Z80 (V)

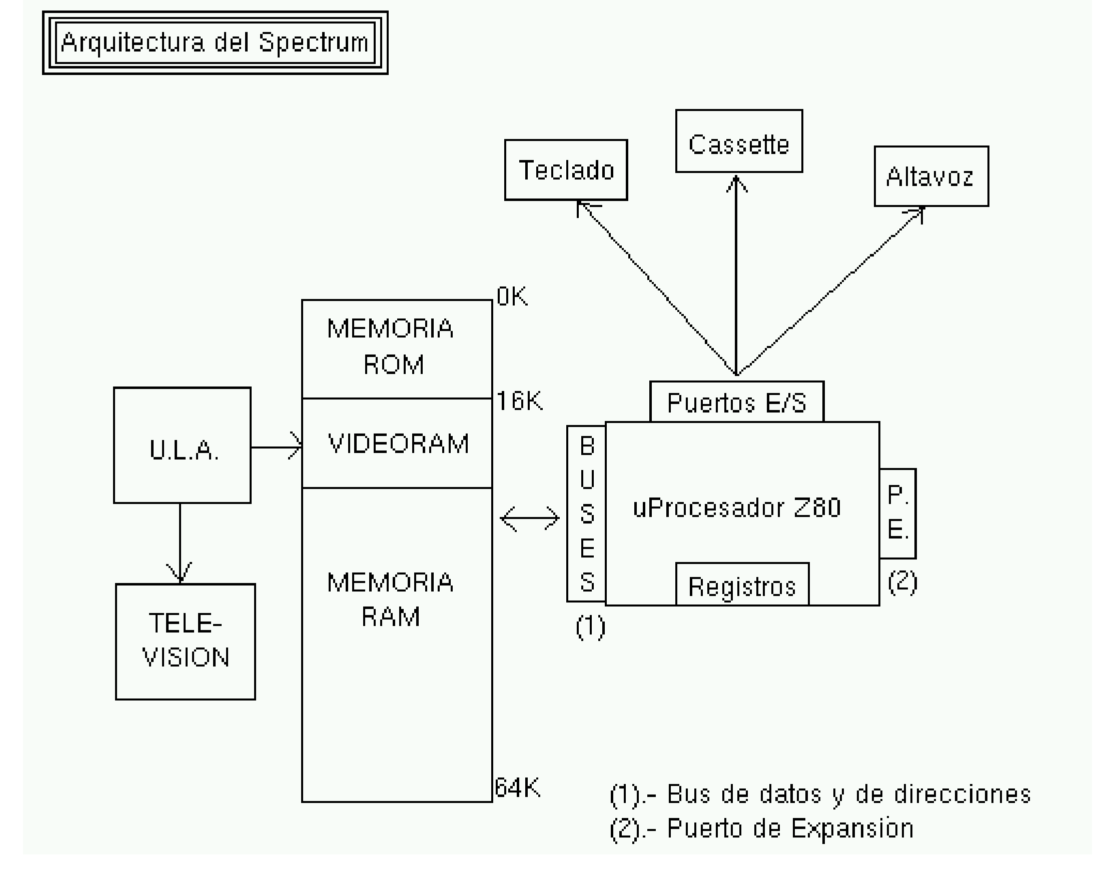
## Puertos de E/S y Tabla de Opcodes

En este capítulo se introducirán las instrucciones **IN** y **OUT** para la exploración de los puertos del microprocesador, mostrando cómo el acceso a dichos puertos nos permitirá la gestión de los diferentes dispositivos conectados al microprocesador (teclado, altavoz, controladora de disco, etc...).

Finalmente, para acabar con la descripción del juego de instrucciones del Z80 veremos algunos ejemplos de *opcodes no documentados*, y una *tabla-resumen* con la mayoría de instrucciones, así como sus tiempos de ejecución y tamaños.

## Los puertos E/S

Como ya vimos en su momento, el microprocesador Z80 se conecta mediante los puertos de entrada/salida de la CPU a los periféricos externos (teclado, cassette y altavoz de audio), pudiendo leer el estado de los mismos (leer del teclado, leer del cassette) y escribir en ellos (escribir en el altavoz para reproducir sonido, escribir en el cassette) por medio de estas conexiones conocidas como "*I/O Ports*".




*Esquema de hardware de un ZX Spectrum*

Aunque para nosotros el teclado o el altavoz puedan ser parte del ordenador, para el Z80, el microprocesador en sí mismo, son tan externos a él como el monitor o el joystick. Nuestro microprocesador accede a todos estos elementos externos mediante una serie de patillas (buses de datos y direcciones) que son conectadas eléctricamente a todos los elementos externos con los que queremos que interactúe. La memoria, el teclado, el altavoz, o los mismos pines del bus trasero del Spectrum, se conectan al Z80 y éste nos permite su acceso a través de dichas líneas, o de los puertos de entrada/salida (I/O).

**IN y OUT**

Ya conocemos la existencia y significado de los puertos y su conexión con el microprocesador. Sólo resta saber: ¿cómo accedemos a un puerto tanto para leer como para escribir desde nuestros programas en ensamblador?

La respuesta la tienen los comandos **IN** y **OUT** del Z80.

Comenzaremos con **IN**, que nos permite leer el valor de un puerto ya sea directamente, o cargado sobre el registro BC:

**IN registro, (C)**

Leemos el puerto "BC" y ponemos su contenido en el registro especificado.

En realidad, pese a que teóricamente el Spectrum sólo tiene acceso a puertos E/S de 8 bits (0-255), para acceder a los puertos, IN r, (C) pone todo el valor de BC en el bus de direcciones.

**IN A, (puerto)**

Leemos el puerto "A*256 + Puerto" y ponemos su contenido en A. En esta ocasión, el Spectrum pone en el bus de direcciones el valor del registro de 16 bits formado por A y (puerto) (en lugar de BC).

Por ejemplo, estas 2 lecturas de puerto (usando los 2 formatos de la instrucción IN vistos anteriormente) son equivalentes:

```
; Forma 1
LD BC, FFFEh
IN A, (C)       ; A = Lectura de puerto FFFEh

; Forma 2
LD A, FFh
IN A, (FEh)     ; A = Lectura de puerto FFFEh
```

Aunque la instrucción de la "Forma 1" hable del puerto C, en realidad el puerto es un valor de 16 bits y se carga en el registro BC.

De la misma forma, podemos escribir un valor en un puerto con sus equivalentes "OUT":

**OUT (puerto), A**

Escribimos en "puerto" (valor de 8 bits) el valor de A.

**OUT (C), registro**

Escribimos en el puerto "C" el valor contenido en "registro" (aunque se pone el valor de BC en el bus de direcciones).

Curiosamente, como se explica en el excelente documento "//The Undocumented Z80 Documented//" (que habla de las funcionalidades y opcodes no documentados del Z80), los puertos del Spectrum son oficialmente de 8 bits (0-255) aunque realmente se pone o bien BC o bien (A*256)+PUERTO en el bus de direcciones, por lo que en el fondo se pueden acceder a todos los 65536 puertos disponibles.

La forma en que estas instrucciones afectan a los flags es la siguiente:

```
                         Flags
   Instrucción       |S Z H P N C|
 ----------------------------------
   IN A, (n)         |- - - - - -|
   IN r, (C)         |* * * P 0 -|
   OUT (C), r        |- - - - - -|
   OUT (n), A        |- - - - - -|
```

Aunque entre los 2 formatos OUT no debería haber ninguna diferencia funcional, cabe destacar que "OUT (N), A" es 1 t-estado o ciclo de reloj más rápida que "OUT (C), A", tardando 11 y 12 t-estados respectivamente.

**Instrucciones de puerto repetitivas e incrementales**

Al igual que LD carga un valor de un origen a un destino, y tiene sus correspondientes instrucciones incrementales (LDI "carga e incrementa", LDD "carga y decrementa") o repetitivas (LDIR "carga, incrementa y repite BC veces", LDDR "carga, decrementa, y repite BC veces"), IN y OUT tienen sus equivalentes incrementales y repetidores.

Así:

- **IND** :
  - Leemos en la dirección de memoria apuntada por HL ([HL]) el valor contenido en el puerto C.
  - Decrementamos HL.
  - Decrementamos B.
- **INI** :
  - Leemos en la dirección de memoria apuntada por HL ([HL]) el valor contenido en el puerto C.
  - Incrementamos HL.
  - Decrementamos B.
- **OUTD** :
  - Escribimos en el puerto C el valor de la dirección de memoria apuntada por HL ([HL]).
  - Decrementamos HL.
  - Decrementamos B.
- **OUTI** :
  - Escribimos en el puerto C el valor de la dirección de memoria apuntada por HL ([HL]).
  - Incrementamos HL.
  - Decrementamos B.

Y sus versiones repetitivas **INDR, INIR, OTDR** y **OTIR**, que realizan la misma función que sus hermanas incrementales, repitiéndolo hasta que BC sea cero.

Las afectaciones de flags de estas funciones son las siguientes:

Flags:

```
                         Flags
   Instrucción       |S Z H P N C|
 ----------------------------------
   INI               |? * ? ? 1 ?|
   IND               |? * ? ? 1 ?|
   OUTI              |? * ? ? 1 ?|
   OUTD              |? * ? ? 1 ?|
   INDR              |? 1 ? ? 1 ?|
   INIR              |? 1 ? ? 1 ?|
   OTDR              |? 1 ? ? 1 ?|
```

```
OTIR              |? 1 ? ? 1 ?|
```

Nota: Pese a que la documentación oficial dice que estas instrucciones no afectan al Carry Flag, las pruebas hechas a posteriori y recopiladas en la información disponible sobre Opcodes No Documentados del Z80 sugieren que sí que son modificados.

**Algunos puertos E/S comunes**

Para terminar con el tema de los puertos de Entrada y Salida, vamos a hacer referencia a algunos puertos disponibles en el Sinclair Spectrum (algunos de ellos sólo en ciertos modelos).

Como veremos en capítulo dedicado al teclado, existe una serie de puertos E/S que acceden directamente a la lectura del estado de las diferentes teclas de nuestro Spectrum. Leyendo del puerto adecuado, y chequeando en la respuesta obtenida el bit concreto asociado a la tecla que queremos consultar podremos conocer si una determinada tecla está pulsada (0) o no pulsada (1), como podemos ver en el siguiente ejemplo:

```
 ; Lectura de la tecla "P" en un bucle
 ORG 50000

bucle:
 LD BC, $DFFE         ; Semifila "P" a "Y"
 IN A, (C)            ; Leemos el puerto
 BIT 0, A             ; Testeamos el bit 0
 JR Z, salir          ; Si esta a 0 (pulsado) salir.
 JR bucle             ; Si no (a 1, no pulsado) repetimos

salir:
 RET
```

El anterior ejemplo lee constantemente el puerto $DFFE a la espera de que el bit 0 de la respuesta obtenida de dicha lectura sea 0, lo que quiere decir que la tecla "p" ha sido pulsada.

Aunque los veremos en su momento en profundidad, estos son los puertos asociados a las diferentes filas de teclas:

| **Puerto** | **Bits:** | **D4** | **D3** | **D2** | **D1** | **D0** |
|---|---|---|---|---|---|---|
| 65278d (FEFEh) | Teclas: | "V" | "C" | "X" | "Z" | CAPS |
| 65022d (FDFEh) | Teclas: | "G" | "F" | "D" | "S" | "A" |
| 64510d (FBFEh) | Teclas: | "T" | "R" | "E" | "W" | "Q" |
| 63486d (F7FEh) | Teclas: | "5" | "4" | "3" | "2" | "1" |
| 61438d (EFFEh) | Teclas: | "0" | "9" | "8" | "7" | "6" |
| 57342d (DFFEh) | Teclas: | "Y" | "U" | "I" | "O" | "P" |
| 49150d (BFFEh) | Teclas: | "H" | "J" | "K" | "L" | ENTER |
| 32766d (7FFEh) | Teclas: | "B" | "N" | "M" | SYMB | SPACE |

El bit 6 de los puertos que hemos visto para el teclado tiene un valor aleatorio, excepto cuando se pulsa PLAY en el cassette, y es a través de dicho bit de donde podremos obtener los datos a cargar.

La escritura en el puerto 00FEh permite *acceder al altavoz* (bit 4) y a la señal de audio para grabar a cinta (bit 3). Los bits 0, 1 y 2 controlan *el color del borde*, como podemos ver en el siguiente ejemplo:

```
 ; Cambio del color del borde al pulsar espacio
 ORG 50000

 LD B, 6              ; 6 iteraciones, color inicial borde

bucle:
 LD A, $7F            ; Semifila B a ESPACIO
 IN A, ($FE)          ; Leemos el puerto
 BIT 0, A             ; Testeamos el bit 0 (ESPACIO)
 JR NZ, bucle         ; Si esta a 1 (no pulsado), esperar

 LD A, B              ; A = B
 OUT (254), A         ; Cambiamos el color del borde
```

```
suelta_tecla:           ; Ahora esperamos a que se suelte la tecla
 LD A, $7F             ; Semifila B a ESPACIO
 IN A, ($FE)           ; Leemos el puerto
 BIT 0, A              ; Testeamos el bit 0
 JR Z, suelta_tecla    ; Saltamos hasta que se suelte

 djnz bucle            ; Repetimos "B" veces

salir:
 RET

 END 50000             ; Ejecucion en 50000
```




*Ejecución del programa de cambio de borde*

El puerto 7FFDh gestiona la *paginación en los modos de 128K*, permitiendo cambiar el modelo de páginas de memoria (algo que no vamos a ver en este capítulo).

Los puertos BFFDh y FFFDh gestionan el *chip de sonido* en aquellos modelos que dispongan de él, así como el RS232/MIDI y el interfaz AUX.

Finalmente, el puerto 0FFDh gestiona *el puerto paralelo de impresora*, y los puertos 2FFDh y 3FFDh permiten gestionar la controladora de disco en aquellos modelos de Spectrum que dispongan de ella.

Podéis encontrar más información sobre los puertos de Entrada y Salida en el *capítulo 8 sección 23 del manual del +2A y +3*, disponible online en World Of Spectrum.

**Tabla de instrucciones, ciclos y tamaños**

A continuación se incluye una tabla donde se hace referencia a las instrucciones del microprocesador Z80 (campo Mnemonic), los ciclos de reloj que tarda en ejecutarse (campo Clck), el tamaño en bytes de la instrucción codificada (Siz), la afectación de Flags (SZHPNC), el opcode y su descripción en cuanto a ejecución.

La tabla forma parte de un documento llamado "**The Complete Z80 OP-Code Reference**", de *Devin Gardner*.

| Mnemonic | Clck | Size | SZHPNC | OP-Code | Description | Notes |
|---|---|---|---|---|---|---|
| ADC A,r | 4 | 1 | ***V0* | 88+rb | Add with Carry | A=A+s+CY |
| ADC A,N | 7 | 2 | | CE XX | | |
| ADC A,(HL) | 7 | 1 | | 8E | | |
| ADC A,(IX+N) | 19 | 3 | | DD 8E XX | | |
| ADC A,(IY+N) | 19 | 3 | | FD 8E XX | | |
| ADC HL,BC | 15 | 2 | **?V0* | ED 4A | Add with Carry | HL=HL+ss+CY |
| ADC HL,DE | 15 | 2 | | ED 5A | | |
| ADC HL,HL | 15 | 2 | | ED 6A | | |
| ADC HL,SP | 15 | 2 | | ED 7A | | |
| ADD A,r | 4 | 1 | ***V0* | 80+rb | Add (8-bit) | A=A+s |

| | | | | | | |
|---|---|---|---|---|---|---|
| ADD A,N | 7 | 2 | | C6 XX | | |
| ADD A,(HL) | 7 | 1 | | 86 | | |
| ADD A,(IX+N) | 19 | 3 | | DD 86 XX | | |
| ADD A,(IY+N) | 19 | 3 | | FD 86 XX | | |
| ADD HL,BC | 11 | 1 | --?-0* | 09 | Add (16-bit) | HL=HL+ss |
| ADD HL,DE | 11 | 1 | | 19 | | |
| ADD HL,HL | 11 | 1 | | 29 | | |
| ADD HL,SP | 11 | 1 | | 39 | | |
| ADD IX,BC | 15 | 2 | --?-0* | DD 09 | Add (IX register) | IX=IX+pp |
| ADD IX,DE | 15 | 2 | | DD 19 | | |
| ADD IX,IX | 15 | 2 | | DD 29 | | |
| ADD IX,SP | 15 | 2 | | DD 39 | | |
| ADD IY,BC | 15 | 2 | --?-0* | FD 09 | Add (IY register) | IY=IY+rr |
| ADD IY,DE | 15 | 2 | | FD 19 | | |
| ADD IY,IY | 15 | 2 | | FD 29 | | |
| ADD IY,SP | 15 | 2 | | FD 39 | | |
| AND r | 4 | 1 | ***P00 | A0+rb | Logical AND | A=A&s |
| AND N | 7 | 2 | | E6 XX | | |
| AND (HL) | 7 | 1 | | A6 | | |
| AND (IX+N) | 19 | 3 | | DD A6 XX | | |
| AND (IY+N) | 19 | 3 | | FD A6 XX | | |
| BIT b,r | 8 | 2 | ?*1?0- | CB 40+8*b+rb | Test Bit | m&{2^b} |
| BIT b,(HL) | 12 | 2 | | CB 46+8*b | | |
| BIT b,(IX+N) | 20 | 4 | | DD CB | | |
| | | | | XX 46+8*b | | |
| BIT b,(IY+N) | 20 | 4 | | FD CB XX | | |
| | | | | XX 46+8*b | | |
| CALL NN | 17 | 3 | ------ | CD XX XX | Unconditional Call | -(SP)=PC,PC=nn |
| CALL C,NN | 17/1 | 3 | ------ | DC XX XX | Conditional Call | If Carry = 1 |
| CALL NC,NN | 17/1 | 3 | | D4 XX XX | | If carry = 0 |
| CALL M,NN | 17/1 | 3 | | FC XX XX | | If Sign = 1 (negative) |
| CALL P,NN | 17/1 | 3 | | F4 XX XX | | If Sign = 0 (positive) |
| CALL Z,NN | 17/1 | 3 | | CC XX XX | | If Zero = 1 (ans.=0) |
| CALL NZ,NN | 17/1 | 3 | | C4 XX XX | | If Zero = 0 (non-zero) |
| CALL PE,NN | 17/1 | 3 | | EC XX XX | | If Parity = 1 (even) |
| CALL PO,NN | 17/1 | 3 | | E4 XX XX | | If Parity = 0 (odd) |
| CCF | 4 | 1 | --?-0* | 3F | Complement Carry Flag | CY=~CY |
| CP r | 4 | 1 | ***V1* | B8+rb | Compare | Compare A-s |
| CP N | 7 | 2 | | FE XX | | |
| CP (HL) | 7 | 1 | | BE | | |
| CP (IX+N) | 19 | 3 | | DD BE XX | | |
| CP (IY+N) | 19 | 3 | | FD BE XX | | |
| CPD | 16 | 2 | ****1- | ED A9 | Compare and Decrement | A-(HL),HL=HL-1,BC=BC-1 |

| | | | | | | |
|---|---|---|---|---|---|---|
| CPDR | 21/1 | 2 | ****1- | ED B9 | Compare, Dec., Repeat | CPD till A=(HL)or BC=0 |
| CPI | 16 | 2 | ****1- | ED A1 | Compare and Increment | A-(HL),HL=HL+1,BC=BC-1 |
| CPIR | 21/1 | 2 | ****1- | ED B1 | Compare, Inc., Repeat | CPI till A=(HL)or BC=0 |
| CPL | 4 | 1 | --1-1- | 2F | Complement | A=~A |
| DAA | 4 | 1 | ***P-* | 27 | Decimal Adjust Acc. | A=BCD format (dec.) |
| DEC A | 4 | 1 | ***V1- | 3D | Decrement (8-bit) | s=s-1 |
| DEC B | 4 | 1 | | 05 | | |
| DEC C | 4 | 1 | | 0D | | |
| DEC D | 4 | 1 | | 15 | | |
| DEC E | 4 | 1 | | 1D | | |
| DEC H | 4 | 1 | | 25 | | |
| DEC L | 4 | 2 | | 2D | | |
| DEC (HL) | 11 | 1 | | 35 | | |
| DEC (IX+N) | 23 | 3 | | DD 35 XX | | |
| DEC (IY+N) | 23 | 3 | | FD 35 XX | | |
| DEC BC | 6 | 1 | ------ | 0B | Decrement (16-bit) | ss=ss-1 |
| DEC DE | 6 | 1 | | 1B | | |
| DEC HL | 6 | 1 | | 2B | | |
| DEC SP | 6 | 1 | | 3B | | |
| DEC IX | 10 | 2 | ------ | DD 2B | Decrement | xx=xx-1 |
| DEC IY | 10 | 2 | | FD 2B | | |
| DI | 4 | 1 | ------ | F3 | Disable Interrupts | |
| DJNZ $+2 | 13/8 | 1 | ------ | 10 | Dec., Jump Non-Zero | B=B-1 till B=0 |
| EI | 4 | 1 | ------ | FB | Enable Interrupts | |
| EX (SP),HL | 19 | 1 | ------ | E3 | Exchange | (SP)<->HL |
| EX (SP),IX | 23 | 2 | ------ | DD E3 | | (SP)<->xx |
| EX (SP),IY | 23 | 2 | | FD E3 | | |
| EX AF,AF' | 4 | 1 | ------ | 08 | | AF<->AF' |
| EX DE,HL | 4 | 1 | ------ | EB | | DE<->HL |
| EXX | 4 | 1 | ------ | D9 | Exchange | qq<->qq' (except AF) |
| HALT | 4 | 1 | ------ | 76 | Halt | |
| IM 0 | 8 | 2 | ------ | ED 46 | Interrupt Mode | (n=0,1,2) |
| IM 1 | 8 | 2 | | ED 56 | | |
| IM 2 | 8 | 2 | | ED 5E | | |
| IN A,(N) | 11 | 2 | ------ | DB XX | Input | A=(n) |
| IN (C) | 12 | 2 | ***P0- | ED 70 | Input* | (Unsupported) |
| IN A,(C) | 12 | 2 | ***P0- | ED 78 | Input | r=(C) |
| IN B,(C) | 12 | 2 | | ED 40 | | |
| IN C,(C) | 12 | 2 | | ED 48 | | |
| IN D,(C) | 12 | 2 | | ED 50 | | |
| IN E,(C) | 12 | 2 | | ED 58 | | |
| IN H,(C) | 12 | 2 | | ED 60 | | |
| IN L,(C) | 12 | 2 | | ED 68 | | |

| | | | | | | |
|---|---|---|---|---|---|---|
| INC A | 4 | 1 | ***V0- | 3C | Increment (8-bit) | r=r+1 |
| INC B | 4 | 1 | | 04 | | |
| INC C | 4 | 1 | | 0C | | |
| INC D | 4 | 1 | | 14 | | |
| INC E | 4 | 1 | | 1C | | |
| INC H | 4 | 1 | | 24 | | |
| INC L | 4 | 1 | | 2C | | |
| INC BC | 6 | 1 | ------ | 03 | Increment (16-bit) | ss=ss+1 |
| INC DE | 6 | 1 | | 13 | | |
| INC HL | 6 | 1 | | 23 | | |
| INC SP | 6 | 1 | | 33 | | |
| INC IX | 10 | 2 | ------ | DD 23 | Increment | xx=xx+1 |
| INC IY | 10 | 2 | | FD 23 | | |
| INC (HL) | 11 | 1 | ***V0- | 34 | Increment (indirect) | (HL)=(HL)+1 |
| INC (IX+N) | 23 | 3 | ***V0- | DD 34 XX | Increment | (xx+d)=(xx+d)+1 |
| INC (IY+N) | 23 | 3 | | FD 34 XX | | |
| IND | 16 | 2 | ?*??1- | ED AA | Input and Decrement | (HL)=(C),HL=HL-1,B=B-1 |
| INDR | 21/1 | 2 | ?1??1- | ED BA | Input, Dec., Repeat | IND till B=0 |
| INI | 16 | 2 | ?*??1- | ED A2 | Input and Increment | (HL)=(C),HL=HL+1,B=B-1 |
| INIR | 21/1 | 2 | ?1??1- | ED B2 | Input, Inc., Repeat | INI till B=0 |
| JP $NN | 10 | 3 | ------ | C3 XX XX | Unconditional Jump | PC=nn |
| JP (HL) | 4 | 1 | ------ | E9 | Unconditional Jump | PC=(HL) |
| JP (IX) | 8 | 2 | ------ | DD E9 | Unconditional Jump | PC=(xx) |
| JP (IY) | 8 | 2 | | FD E9 | | |
| JP C,$NN | 10/1 | 3 | ------ | DA XX XX | Conditional Jump | If Carry = 1 |
| JP NC,$NN | 10/1 | 3 | | D2 XX XX | | If Carry = 0 |
| JP M,$NN | 10/1 | 3 | | FA XX XX | | If Sign = 1 (negative) |
| JP P,$NN | 10/1 | 3 | | F2 XX XX | | If Sign = 0 (positive) |
| JP Z,$NN | 10/1 | 3 | | CA XX XX | | If Zero = 1 (ans.= 0) |
| JP NZ,$NN | 10/1 | 3 | | C2 XX XX | | If Zero = 0 (non-zero) |
| JP PE,$NN | 10/1 | 3 | | EA XX XX | | If Parity = 1 (even) |
| JP PO,$NN | 10/1 | 3 | | E2 XX XX | | If Parity = 0 (odd) |
| JR $N+2 | 12 | 2 | ------ | 18 XX | Relative Jump | PC=PC+e |
| JR C,$N+2 | 12/7 | 2 | ------ | 38 XX | Cond. Relative Jump | If cc JR(cc=C,NC,NZ,Z) |
| JR NC,$N+2 | 12/7 | 2 | | 30 XX | | |
| JR Z,$N+2 | 12/7 | 2 | | 28 XX | | |
| JR NZ,$N+2 | 12/7 | 2 | | 20 XX | | |
| LD I,A | 9 | 2 | ------ | ED 47 | Load* | dst=src |
| LD R,A | 9 | 2 | | ED 4F | | |
| LD A,I | 9 | 2 | **0*0- | ED 57 | Load* | dst=src |
| LD A,R | 9 | 2 | | ED 5F | | |
| LD A,r | 4 | 1 | ------ | 78+rb | Load (8-bit) | dst=src |
| LD A,N | 7 | 2 | | 3E XX | | |

| | | | | | | |
|---|---|---|---|---|---|---|
| LD A,(BC) | 7 | 1 | | 0A | | |
| LD A,(DE) | 7 | 1 | | 1A | | |
| LD A,(HL) | 7 | 1 | | 7E | | |
| LD A,(IX+N) | 19 | 3 | | DD 7E XX | | |
| LD A,(IY+N) | 19 | 3 | | FD 7E XX | | |
| LD A,(NN) | 13 | 3 | | 3A XX XX | | |
| LD B,r | 4 | 1 | | 40+rb | | |
| LD B,N | 7 | 2 | | 06 XX | | |
| LD B,(HL) | 7 | 1 | | 46 | | |
| LD B,(IX+N) | 19 | 3 | | DD 46 XX | | |
| LD B,(IY+N) | 19 | 3 | | FD 46 XX | | |
| LD C,r | 4 | 1 | | 48+rb | | |
| LD C,N | 7 | 2 | | 0E XX | | |
| LD C,(HL) | 7 | 1 | | 4E | | |
| LD C,(IX+N) | 19 | 3 | | DD 4E XX | | |
| LD C,(IY+N) | 19 | 3 | | FD 4E XX | | |
| LD D,r | 4 | 1 | | 50+rb | | |
| LD D,N | 7 | 2 | | 16 XX | | |
| LD D,(HL) | 7 | 1 | | 56 | | |
| LD D,(IX+N) | 19 | 3 | | DD 56 XX | | |
| LD D,(IY+N) | 19 | 3 | | FD 56 XX | | |
| LD E,r | 4 | 1 | | 58+rb | | |
| LD E,N | 7 | 2 | | 1E XX | | |
| LD E,(HL) | 7 | 1 | | 5E | | |
| LD E,(IX+N) | 19 | 3 | | DD 5E XX | | |
| LD E,(IY+N) | 19 | 3 | | FD 5E XX | | |
| LD H,r | 4 | 1 | | 60+rb | | |
| LD H,N | 7 | 2 | | 26 XX | | |
| LD H,(HL) | 7 | 1 | | 66 | | |
| LD H,(IX+N) | 19 | 3 | | DD 66 XX | | |
| LD H,(IY+N) | 19 | 3 | | FD 66 XX | | |
| LD L,r | 4 | 1 | | 68+rb | | |
| LD L,N | 7 | 2 | | 2E XX | | |
| LD L,(HL) | 7 | 1 | | 6E | | |
| LD L,(IX+N) | 19 | 3 | | DD 6E XX | | |
| LD L,(IY+N) | 19 | 3 | | FD 6E XX | | |
| LD BC,(NN) | 20 | 4 | ------ | ED 4B XX XX | Load (16-bit) | dst=src |
| LD BC,NN | 10 | 3 | | 01 XX XX | | |
| LD DE,(NN) | 20 | 4 | | ED 5B XX XX | | |
| LD DE,NN | 10 | 3 | | 11 XX XX | | |
| LD HL,(NN) | 20 | 3 | | 2A XX XX | | |
| LD HL,NN | 10 | 3 | | 21 XX XX | | |
| LD SP,(NN) | 20 | 4 | | ED 7B XX XX | | |

| | | | | | | |
|---|---|---|---|---|---|---|
| LD SP,HL | 6 | 1 | | F9 | | |
| LD SP,IX | 10 | 2 | | DD F9 | | |
| LD SP,IY | 10 | 2 | | FD F9 | | |
| LD SP,NN | 10 | 3 | | 31 XX XX | | |
| LD IX,(NN) | 20 | 4 | | DD 2A XX XX | | |
| LD IX,NN | 14 | 4 | | DD 21 XX XX | | |
| LD IY,(NN) | 20 | 4 | | FD 2A XX XX | | |
| LD IY,NN | 14 | 4 | | FD 21 XX XX | | |
| LD (HL),r | 7 | 1 | ------ | 70+rb | Load (Indirect) | dst=src |
| LD (HL),N | 10 | 2 | | 36 XX | | |
| LD (BC),A | 7 | 1 | | 02 | | |
| LD (DE),A | 7 | 1 | | 12 | | |
| LD (NN),A | 13 | 3 | | 32 XX XX | | |
| LD (NN),BC | 20 | 4 | | ED 43 XX XX | | |
| LD (NN),DE | 20 | 4 | | ED 53 XX XX | | |
| LD (NN),HL | 16 | 3 | | 22 XX XX | | |
| LD (NN),IX | 20 | 4 | | DD 22 XX XX | | |
| LD (NN),IY | 20 | 4 | | FD 22 XX XX | | |
| LD (NN),SP | 20 | 4 | | ED 73 XX XX | | |
| LD (IX+N),r | 19 | 3 | | DD 70+rb XX | | |
| LD (IX+N),N | 19 | 4 | | DD 36 XX XX | | |
| LD (IY+N),r | 19 | 3 | | FD 70+rb XX | | |
| LD (IY+N),N | 19 | 4 | | FD 36 XX XX | | |
| LDD | 16 | 2 | --0*0- | ED A8 | Load and Decrement | (DE)=(HL),HL=HL-1,# |
| LDDR | 21/1 | 2 | --000- | ED B8 | Load, Dec., Repeat | LDD till BC=0 |
| LDI | 16 | 2 | --0*0- | ED A0 | Load and Increment | (DE)=(HL),HL=HL+1,# |
| LDIR | 21/1 | 2 | --000- | ED B0 | Load, Inc., Repeat | LDI till BC=0 |
| NEG | 8 | 2 | ***V1* | ED 44 | Negate | A=-A |
| NOP | 4 | 1 | ------ | 00 | No Operation | |
| OR r | 4 | 1 | ***P00 | B0+rb | Logical inclusive OR | A=Avs |
| OR N | 7 | 2 | | F6 XX | | |
| OR (HL) | 7 | 1 | | B6 | | |
| OR (IX+N) | 19 | 3 | | DD B6 XX | | |
| OR (IY+N) | 19 | 3 | | FD B6 XX | | |
| OUT (N),A | 11 | 2 | ------ | D3 XX | Output | (n)=A |
| OUT (C),0 | 12 | 2 | ------ | ED 71 | Output* | (Unsupported) |
| OUT (C),A | 12 | 2 | ------ | ED 79 | Output | (C)=r |
| OUT (C),B | 12 | 2 | | ED 41 | | |
| OUT (C),C | 12 | 2 | | ED 49 | | |
| OUT (C),D | 12 | 2 | | ED 51 | | |
| OUT (C),E | 12 | 2 | | ED 59 | | |
| OUT (C),H | 12 | 2 | | ED 61 | | |
| OUT (C),L | 12 | 2 | | ED 69 | | |

| | | | | | | |
|---|---|---|---|---|---|---|
| OUTD | 16 | 2 | ?*??1- | ED AB | Output and Decrement | (C)=(HL),HL=HL-1,B=B-1 |
| OTDR | 21/1 | 2 | ?1??1- | ED BB | Output, Dec., Repeat | OUTD till B=0 |
| OUTI | 16 | 2 | ?*??1- | ED A3 | Output and Increment | (C)=(HL),HL=HL+1,B=B-1 |
| OTIR | 21/1 | 2 | ?1??1- | ED B3 | Output, Inc., Repeat | OUTI till B=0 |
| POP AF | 10 | 1 | ------ | F1 | Pop | qq=(SP)+ |
| POP BC | 10 | 1 | | C1 | | |
| POP DE | 10 | 1 | | D1 | | |
| POP HL | 10 | 1 | | E1 | | |
| POP IX | 14 | 2 | ------ | DD E1 | Pop | xx=(SP)+ |
| POP IY | 14 | 2 | | FD E1 | | |
| PUSH AF | 11 | 1 | ------ | F5 | Push | -(SP)=qq |
| PUSH BC | 11 | 1 | | C5 | | |
| PUSH DE | 11 | 1 | | D5 | | |
| PUSH HL | 11 | 1 | | E5 | | |
| PUSH IX | 15 | 2 | ------ | DD E5 | Push | -(SP)=xx |
| PUSH IY | 15 | 2 | | FD E5 | | |
| RES b,r | 8 | 2 | ------ | CB 80+8*b+rb | Reset bit | m=m&{~2^b} |
| RES b,(HL) | 15 | 2 | ------ | CB 86+8*b | | |
| RES b,(IX+N) | 23 | 4 | ------ | DD CB<br>XX 86+8*b | | |
| RES b,(IY+N) | 23 | 4 | ------ | FD CB<br>XX 86+8*b | | |
| RET | 10 | 1 | ------ | C9 | Return | PC=(SP)+ |
| RET C | 11/5 | 1 | ------ | D8 | Conditional Return | If Carry = 1 |
| RET NC | 11/5 | 1 | | D0 | | If Carry = 0 |
| RET M | 11/5 | 1 | | F8 | | If Sign = 1 (negative) |
| RET P | 11/5 | 1 | | F0 | | If Sign = 0 (positive) |
| RET Z | 11/5 | 1 | | C8 | | If Zero = 1 (ans.=0) |
| RET NZ | 11/5 | 1 | | C0 | | If Zero = 0 (non-zero) |
| RET PE | 11/5 | 1 | | E8 | | If Parity = 1 (even) |
| RET PO | 11/5 | 1 | | E0 | | If Parity = 0 (odd) |
| RETI | 14 | 2 | ------ | ED 4D | Return from Interrupt | PC=(SP)+ |
| RETN | 14 | 2 | ------ | ED 45 | Return from NMI | PC=(SP)+ |
| RLA | 4 | 1 | --0-0* | 17 | Rotate Left Acc. | A={CY,A}<- |
| RL r | 8 | 2 | **0P0* | CB 10+rb | Rotate Left | m={CY,m}<- |
| RL (HL) | 15 | 2 | | CB 16 | | |
| RL (IX+N) | 23 | 4 | | DD CB XX 16 | | |
| RL (IY+N) | 23 | 4 | | FD CB XX 16 | | |
| RLCA | 4 | 1 | --0-0* | 07 | Rotate Left Cir. Acc. | A=A<- |
| RLC r | 8 | 2 | **0P0* | CB 00+rb | Rotate Left Circular | m=m<- |
| RLC (HL) | 15 | 2 | | CB 06 | | |
| RLC (IX+N) | 23 | 4 | | DD CB XX 06 | | |
| RLC (IY+N) | 23 | 4 | | FD CB XX 06 | | |

| | | | | | | |
|---|---|---|---|---|---|---|
| RLD | 18 | 2 | **0P0- | ED 6F | Rotate Left 4 bits | {A,(HL)}={A,(HL)}<- ## |
| RRA | 4 | 1 | --0-0* | 1F | Rotate Right Acc. | A=->{CY,A} |
| RR r | 8 | 2 | **0P0* | CB 18+rb | Rotate Right | m=->{CY,m} |
| RR (HL) | 15 | 2 | | CB 1E | | |
| RR (IX+N) | 23 | 4 | | DD CB XX 1E | | |
| RR (IY+N) | 23 | 4 | | FD CB XX 1E | | |
| RRCA | 4 | 1 | --0-0* | 0F | Rotate Right Cir.Acc. | A=->A |
| RRC r | 8 | 2 | **0P0* | CB 08+rb | Rotate Right Circular | m=->m |
| RRC (HL) | 15 | 2 | | CB 0E | | |
| RRC (IX+N) | 23 | 4 | | DD CB XX 0E | | |
| RRC (IY+N) | 23 | 4 | | FD CB XX 0E | | |
| RRD | 18 | 2 | **0P0- | ED 67 | Rotate Right 4 bits | {A,(HL)}=->{A,(HL)} ## |
| RST 0 | 11 | 1 | ------ | C7 | Restart | (p=0H,8H,10H,...,38H) |
| RST 08H | 11 | 1 | | CF | | |
| RST 10H | 11 | 1 | | D7 | | |
| RST 18H | 11 | 1 | | DF | | |
| RST 20H | 11 | 1 | | E7 | | |
| RST 28H | 11 | 1 | | EF | | |
| RST 30H | 11 | 1 | | F7 | | |
| RST 38H | 11 | 1 | | FF | | |
| SBC r | 4 | 1 | ***V1* | 98+rb | Subtract with Carry | A=A-s-CY |
| SBC A,N | 7 | 2 | | DE XX | | |
| SBC (HL) | 7 | 1 | | 9E | | |
| SBC A,(IX+N) | 19 | 3 | | DD 9E XX | | |
| SBC A,(IY+N) | 19 | 3 | | FD 9E XX | | |
| SBC HL,BC | 15 | 2 | **?V1* | ED 42 | Subtract with Carry | HL=HL-ss-CY |
| SBC HL,DE | 15 | 2 | | ED 52 | | |
| SBC HL,HL | 15 | 2 | | ED 62 | | |
| SBC HL,SP | 15 | 2 | | ED 72 | | |
| SCF | 4 | 1 | --0-01 | 37 | Set Carry Flag | CY=1 |
| SET b,r | 8 | 2 | ------ | CB C0+8*b+rb | Set bit | m=mv{2^b} |
| SET b,(HL) | 15 | 2 | | CB C6+8*b | | |
| SET b,(IX+N) | 23 | 4 | | DD CB XX C6+8*b | | |
| SET b,(IY+N) | 23 | 4 | | FD CB XX C6+8*b | | |
| SLA r | 8 | 2 | **0P0* | CB 20+rb | Shift Left Arithmetic | m=m*2 |
| SLA (HL) | 15 | 2 | | CB 26 | | |
| SLA (IX+N) | 23 | 4 | | DD CB XX 26 | | |
| SLA (IY+N) | 23 | 4 | | FD CB XX 26 | | |
| SRA r | 8 | 2 | **0P0* | CB 28+rb | Shift Right Arith. | m=m/2 |
| SRA (HL) | 15 | 2 | | CB 2E | | |
| SRA (IX+N) | 23 | 4 | | DD CB XX 2E | | |

| | | | | | | |
|---|---|---|---|---|---|---|
| SRA (IY+N) | 23 | 4 | | FD CB XX 2E | | |
| SLL r | 8 | 2 | **0P0* | CB 30+rb | Shift Left Logical* | m={0,m,CY}<- |
| SLL (HL) | 15 | 2 | | CB 36 | | (SLL instructions |
| SLL (IX+N) | 23 | 4 | | DD CB XX 36 | | are Unsupported) |
| SLL (IY+N) | 23 | 4 | | FD CB XX 36 | | |
| SRL r | 8 | 2 | **0P0* | CB 38+rb | Shift Right Logical | m=->{0,m,CY} |
| SRL (HL) | 15 | 2 | | CB 3E | | |
| SRL (IX+N) | 23 | 4 | | DD CB XX 3E | | |
| SRL (IY+N) | 23 | 4 | | FD CB XX 3E | | |
| SUB r | 4 | 1 | ***V1* | 90+rb | Subtract | A=A-s |
| SUB N | 7 | 2 | | D6 XX | | |
| SUB (HL) | 7 | 1 | | 96 | | |
| SUB (IX+N) | 19 | 3 | | DD 96 XX | | |
| SUB (IY+N) | 19 | 3 | | FD 96 XX | | |
| XOR r | 4 | 1 | ***P00 | A8+rb | Logical Exclusive OR | A=Axs |
| XOR N | 7 | 2 | | EE XX | | |
| XOR (HL) | 7 | 1 | | AE | | |
| XOR (IX+N) | 19 | 3 | | DD AE XX | | |
| XOR (IY+N) | 19 | 3 | | FD AE XX | | |

La leyenda para interpretar esta tabla es la siguiente:

| **Símbolo** | **Significado** |
|---|---|
| n | Immediate addressing |
| nn | Immediate extended addressing |
| e | Relative addressing (PC=PC+2+offset) |
| (nn) | Extended addressing |
| (xx+d) | Indexed addressing |
| r | Register addressing |
| (rr) | Register indirect addressing |
| | Implied addressing |
| b | Bit addressing |
| p | Modified page zero addressing (see RST) |
| * | Undocumented opcode |
| A B C D E | Registers (8-bit) |
| AF BC DE HL | Register pairs (16-bit) |
| F | Flag register (8-bit) |
| I | Interrupt page address register (8-bit) |
| IX IY | Index registers (16-bit) |
| PC | Program Counter register (16-bit) |
| R | Memory Refresh register |
| SP | Stack Pointer register (16-bit) |
| b | One bit (0 to 7) |
| cc | Condition (C,M,NC,NZ,P,PE,PO,Z) |

| | |
|---|---|
| d | One-byte expression (-128 to +127) |
| dst | Destination s, ss, (BC), (DE), (HL), (nn) |
| e | One-byte expression (-126 to +129) |
| m | Any register r, (HL) or (xx+d) |
| n | One-byte expression (0 to 255) |
| nn | Two-byte expression (0 to 65535) |
| pp | Register pair BC, DE, IX or SP |
| qq | Register pair AF, BC, DE or HL |
| qq' | Alternative register pair AF, BC, DE or HL |
| r | Register A, B, C, D, E, H or L |
| rr | Register pair BC, DE, IY or SP |
| s | Any register r, value n, (HL) or (xx+d) |
| src | Source s, ss, (BC), (DE), (HL), nn, (nn) |
| ss | Register pair BC, DE, HL or SP |
| xx | Index register IX or IY |
| + - * / ^ | Add/subtract/multiply/divide/exponent |
| & ~ v x | Logical AND/NOT/inclusive OR/exclusive OR |
| <- -> | Rotate left/right |
| ( ) | Indirect addressing |
| ( )+ -( ) | Indirect addressing auto-increment/decrement |
| { } | Combination of operands |
| # | Also BC=BC-1,DE=DE-1 |
| ## | Only lower 4 bits of accumulator A used |

Unos apuntes sobre esta tabla:

1.- En instrucciones como "ADC A, r" podemos ver una defición del OPCODE como "88+rb".

En este caso, el opcode final se obtendría sumando a "88h" un valor de 0 a 7 según el registro al que nos referimos:

| **Registro** | **Valor RB** |
|---|---|
| A | 7 |
| B | 0 |
| C | 1 |
| D | 2 |
| E | 3 |
| H | 4 |
| L | 5 |
| (HL) | 6 |

Por ejemplo, "ADC A, B" se codificaría en memoria como "88+0=88".

2.- En los saltos hay 2 tiempos de ejecución diferentes (por ejemplo, 10/1). En este caso el valor más alto (10) son los t-estados o ciclos que toma la instrucción cuando el salto se realiza, y el más bajo (1) es lo que tarda la instrucción cuando no se salta al destino. Como véis, a la hora de programar una rutina que tenga saltos o bifurcaciones, es interesante programarla de forma que el caso más común, el que se produzca la mayoría de las veces, no produzca un salto.

3.- La descripción de las afectaciones de flags son las siguientes:

```
--------+-------+--------------------------------------------
```

```
| F      | -*01? |Flag unaffected/affected/reset/set/unknown   |
| S      | S     |Sign flag (Bit 7)                             |
| Z      |  Z    |Zero flag (Bit 6)                             |
| HC     |   H   |Half Carry flag (Bit 4)                       |
| P/V    |    P  |Parity/Overflow flag (Bit 2, V=overflow)      |
| N      |     N |Add/Subtract flag (Bit 1)                     |
| CY     |      C|Carry flag (Bit 0)                            |
+---------------+----------------------------------------------+
```

**Instrucciones no documentadas del Z80**

En Internet podemos encontrar gran cantidad de documentación acerca del Z80 y su juego de instrucciones, incluyendo las especificaciones oficiales del microprocesador Z80 de Zilog.

No obstante, existen una serie de instrucciones u opcodes que el microprocesador puede ejecutar y que no están detallados en la documentación oficial de Zilog.

Con respecto a esto, tenemos la suerte de disponer de algo que los programadores de la época del Spectrum no tenían: una descripción detallada de las instrucciones no documentadas del Z80. Aunque la mayoría son instrucciones repetidas de sus versiones documentadas, hay algunas instrucciones curiosas y a las que tal vez le podamos sacar alguna utilidad.

¿Por qué existen estos opcodes y no fueron documentados? Supongo que algunos de ellos no fueron considerados como "merecedores de utilidad alguna" y los ingenieros de Zilog no los documentaron, o tal vez sean simplemente un resultado no previsto de la ejecución del Z80 porque los diseñadores no pensaron que al microprocesador pudieran llegarle dichos códigos. El caso es que para el microprocesador existen "todos" los opcodes, otra cosa es qué haga al leerlos y decodificarlos. En este caso algunos de ellos realizan funciones válidas mientras que otros son el equivalente a ejecutar 2 instrucciones NOP, por ejemplo.

¿Cuál es la utilidad de estas instrucciones para los programadores? Para ser sinceros, como programadores con un ensamblador o un ensamblador cruzado, poca.

Si haces tus programas desde cero con un programa ensamblador, éste se encargará de la conversión de instrucciones estándar a opcodes, aunque no viene mal conocer la existencia de estas instrucciones. Para los programadores de emuladores y de desensambladores, el conocimiento de estos opcodes es vital.

El juego Sabre Wulf, por ejemplo, utiliza una de estas instrucciones en la determinación del camino de uno de los enemigos en pantalla (la instrucción SLL, que veremos a continuación), hasta el punto en que los primeros emuladores de Spectrum emulaban mal este juego hasta que incluyeron dicha instrucción en la emulación.

Los "*undocumented opcodes*" son esencialmente opcodes con prefijos CB, ED, DD o FD que hacen unas determinadas operaciones y que no están incluídos en la "lista oficial" que hemos visto hasta ahora. Todos los ejemplos que veremos a continuación están extraídos del documento "*The Undocumented Z80 Documented*", de Sean Young.

**Prefijo CB**

Por ejemplo, los opcodes CB 30, CB 31, CB 32, CB 33, CB 34, CB 35, CB 36 y CB 37 definen una nueva instrucción: **SLL**.

| OPCODE | INSTRUCCION |
|---|---|
| CB 30 | SLL B |
| CB 31 | SLL C |
| CB 32 | SLL D |
| CB 33 | SLL E |
| CB 34 | SLL H |
| CB 35 | SLL L |
| CB 36 | SLL (HL) |
| CB 37 | SLL A |

**SLL (Shift Logical Left)** funciona exactamente igual que SLA salvo porque pone a 1 el bit 0 (mientras que SLA lo ponía a 0).

**Prefijos DD y FD**

En general, una instrucción precedida por el opcode DD se ejecuta igual que sin él excepto por las siguientes reglas:

- Si la instrucción usaba el registro HL, éste se sustituye por IX (excepto en las instrucciones EX DE, HL y

EXX).

- Cualquier uso de (HL) se reemplaza por (IX+d), excepto JP (HL).
- Cualquier acceso a H se reemplaza por IXh (byte alto de IX), excepto en el uso de (IX+d).
- Cualquier acceso a L se reemplaza por IXl (byte alto de IX), excepto en el uso de (IX+d).

Por ejemplo:

| Sin el prefijo DD | Con el Prefijo DD |
|---|---|
| LD HL, 0 | LD IX, 0 |
| LD H, A | LD IXh, A |
| LD H, (HL) | LD H, (IX+d) |

El caso de FD es exactamente igual que el de DD, pero usando el registro IY en lugar del IX.

**Prefijo ED**

Hay una gran cantidad de instrucciones ED XX indocumentadas. Muchos de ellos realizan la misma función que sus equivalentes sin ED delante, mientras que otros simplemente son leídos y decodificados, resultando, a niveles prácticos, equivalentes a 2 instrucciones NOP. Veamos algunos de ellos:

| OPCODE | Instrucción |
|---|---|
| ED 4C | NEG |
| ED 4E | IM 0 |
| ED 44 | NEG |
| ED 45 | RETN |
| ED 5C | NEG |
| ED 5D | RETN |
| ED 64 | NEG |
| ED 65 | RETN |
| ED 66 | IM 0 |
| ED 6C | NEG |
| ED 6D | RETN |
| ED 6E | IM 0 |
| ED 70 | IN (C) / IN F,(C) |
| ED 71 | OUT (C),0 |
| ED 74 | NEG |
| ED 75 | RETN |
| ED 76 | IM1 |
| ED 77 | NOP |
| ED 7C | NEG |
| ED 7D | RETN |
| ED 7E | IM2 |
| ED 7F | NOP |

Aparte de los duplicados de NOP, NEG, IM0, etc, podemos ver un par de instrucciones curiosas y que nos pueden ser de utilidad. Por ejemplo:

```
ED 70       IN (C)
```

Esta instrucción lee el puerto C, pero no almacena el resultado de la lectura en ningún lugar. No obstante, altera los flags del registro F como corresponde al resultado leído. Puede ser interesante si sólo nos interesa, por ejemplo, si el valor leído es cero o no (flag Z), y no queremos perder un registro para almacenar el resultado.

**Prefijos DDCB y FDCB**

Las instrucciones DDCB y FDCB no documentadas almacenan el resultado de la operación de la instrucción equivalente sin prefijo (si existe dicho resultado) en uno de los registros de propósito general: B, C, D, E, H, L, ninguno o A, según los 3 bits más bajos del último byte del opcode (000=B, 001=C, 010=D, etc).

Así, supongamos el siguiente opcode sí documentado:

```
DD CB 01 06         RLC (IX+01h)
```

Si hacemos los 3 últimos bits de dicho opcode 010 (010), el resultado de la operación se copia al registro D (010 = D en nuestra definición anterior), con lo que realmente, en lugar de "RLC (IX+01h)" se ejecuta:

```
LD D, (IX+01h)
RLC D
LD (IX+01h), D
```

La notación que sugiere *Sean Young* para estos opcodes es: "RLC (IX+01h), D".

Con el prefijo FDCB ocurre igual que con DDCB, salvo que se usa el registro IY en lugar de IX.

**De la teoria a la practica**

Con este capítulo hemos cubierto el 99% de las instrucciones soportadas por el microprocesador Z80. Con la excepción de los *Modos de Interrupciones* del Z80 y sus aplicaciones, ya tenemos a nuestra disposición las piezas básicas para formar cualquier programa o rutina en ensamblador.

No obstante, todavía quedan por delante muchas horas de programación para dominar este lenguaje, así como diferentes técnicas, trucos, rutinas y mapas de memoria que nos permitan dibujar nuestros gráficos, realizar rutinas complejas, utilizar el sistema de interrupciones del microprocesador para realizar controles de temporización de nuestros programas, o reproducir sonido.

**FICHEROS**

- Ejemplo de reset por el mal uso de la pila.
- Fichero tap del ejemplo reset.asm.
- Experimentando con CALL NZ.
- Fichero tap del ejemplo call_nz.asm.
- Sencillo fundido de pantalla.
- Fichero tap del ejemplo fade.asm.
- Cambiando el borde (ASM)..
- Fichero tap del ejemplo borde.asm.

**LINKS**

- Set de caracteres.
- Web del Z80.
- Z80 Reference de WOS.
- Z80 Reference de TI86.
- FAQ de Icarus Productions.
- Microfichas de CM de MicroHobby.
- Curso de ASM de z80.info.
- Ensamblador PASMO.
- Manual +3: Puertos E/S.
- Manual +3: Set de caracteres.
- Tablas de ensamblado y t-estados (pulsar en z80.txt, z80_reference.txt, z80time.txt).
- Instrucciones no documentadas.
- Instrucciones no documentadas.

Santiago Romero

# Paginación de memoria 128K

Si el bus de direcciones del Spectrum es de 16 bits y por lo tanto sólo puede acceder a posiciones de memoria entre 0 y 65535¿ ¿cómo se las arreglan los modelos de 128KB para acceder a la memoria situada entre las celdillas 65535 y 131071? La respuesta es un mecanismo tan simple como ingenioso: se utiliza una técnica conocida como paginación de memoria.

En este artículo aprenderemos a aprovechar los 128K de memoria de que disponen los modelos de Spectrum 128K, +2, +2A y +3. Gracias a la paginación nos saltaremos la limitación de direccionamiento de 64K del microprocesador Z80 para acceder a la totalidad de la memoria disponible.

**Paginación**

Los modelos de Spectrum 128K, +2, +2A, +2B y +3 disponen de 128KB de memoria, aunque no toda está disponible simultáneamente. Al igual que en el modelo de 48KB, el microprocesador Z80 sólo puede direccionar 64K-direcciones de memoria, por lo que para acceder a esta memoria "extra", se divide en bloques de 16KB y se "mapea" (pagina) sobre la dirección de memoria 0xc000.

¿Qué quiere decir esto? Que nuestro procesador, con un bus de direcciones de 16 bits, sólo puede acceder a la memoria para leer las "celdillas" entre 0×0000 y 0xFFFF (0-65535). Para leer casillas de memoria superiores a 65535, harían falta más de 16 bits de direcciones, ya que 65535 es el mayor entero que se puede formar con 16 bits (1111111111111111b).

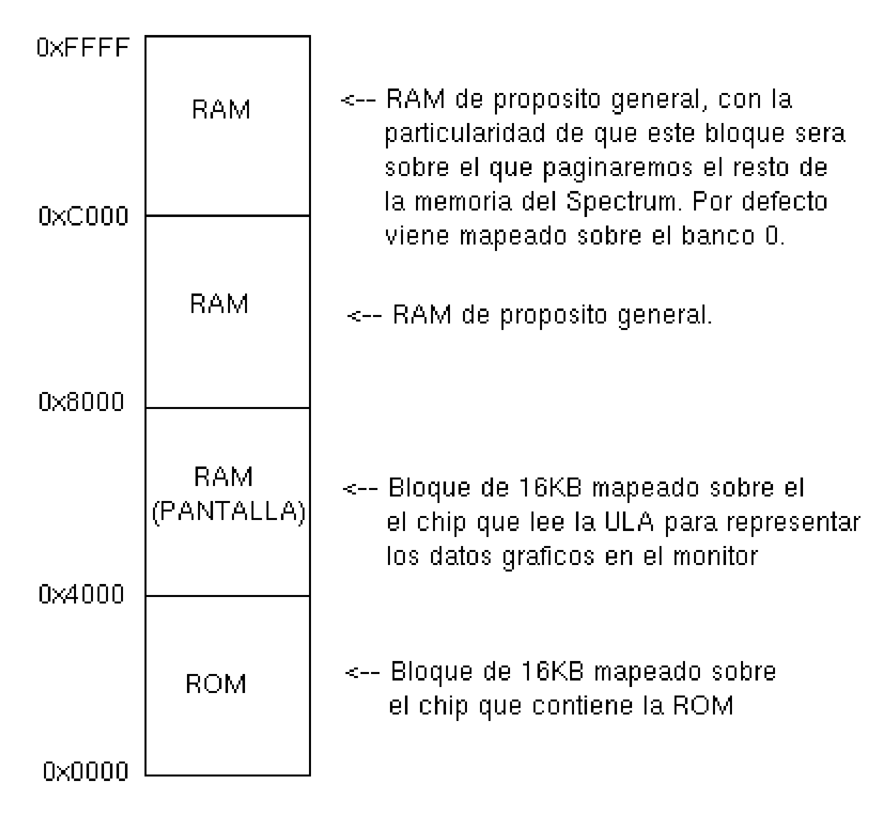
Así que ¿ ¿cómo hacemos para utilizar más de 64KB de memoria si nuestro procesador sólo puede leer datos de la celdilla 0, 1, 2, 3 ¿ 65534 y 65535? La respuesta es: **mediante la paginación**.

Los 64KB de memoria del Spectrum, se dividen en 4 bloques de 16KB. El primer bloque (0×0000 - 0×4000) está mapeado sobre la ROM del Spectrum. Accediendo a los bytes desde 0×0000 a 0x3FFF de la memoria estamos accediendo a los bytes 0×0000 a 0x3FFF del "chip (físico)" de 16K de memoria que almacena la ROM del Spectrum. Se dice, pues, que la ROM está mapeada (o paginada) sobre 0×0000 en el mapa de memoria.

Dejando la ROM de lado (16KB), los 48KB de memoria restantes están formados por 3 bloques de 16KB. El segundo bloque de 16K de memoria (primero de los 3 que estamos comentando), en el rango 0×4000 a 0x7FFF está mapeado sobre el "chip de memoria" que almacena el área de pantalla del Spectrum. El tercero (0×8000 a 0xBFFFF) es memoria de propósito general (normalmente cargaremos nuestros programas aquí).

El bloque que nos interesa a nosotros es el cuarto. La zona final del área de memoria, desde 0xC000 a 0xFFFF, es nuestra "ventana" hacia el resto de memoria del 128K. Dividiendo los 128KB de memoria en bloques de 16KB, tendremos 8 bloques que podremos "montar" (o paginar) sobre el área 0xC000 a 0xFFFF.

Veamos una figura donde se muestra el estado del **mapa de memoria del Spectrum**:




*Mapa de memoria del Spectrum*

Cualquiera de los 8 bloques de 16KB (128 KB / 16 KB = 8 bloques) puede ser "mapeado" en los 16Kb que van desde 0xc000 a 0xffff, y podremos cambiar este mapeo de un bloque a otro mediante instrucciones I/O concretas.

La última porción de 16KB de la memoria es, pues, una "ventana" que podemos deslizar para que nos dé acceso a cualquiera de los 8 bloques disponibles. Esto nos permite "paginar" el bloque 0 y escribir o leer sobre él, por ejemplo. El byte 0 del bloque 0 se accede a través de la posición de memoria 0xC000 una vez paginado, el byte 1 desde 0xC0001, y así hasta el byte 16383, al que accedemos mediante 0xFFFF.

Si paginamos el bloque 1 en nuestra ventana 0xC000-0xFFFF, cuando accedamos a este rango de memoria, ya no accederíamos a los mismos datos que guardamos en el banco 0, sino a la zona de memoria "Banco 1". Es posible incluso mapear la zona de pantalla (Banco 5), de forma que las direcciones 0×4000 y 0xC000 serían equivalentes: los 8 primeros píxeles de la pantalla.

El mapa de memoria del Spectrum con los bloques mapeables/paginables sobre 0xC000 es el siguiente:

| 0xFFFF | Banco 0 | Banco 1 | Banco 2 [1] | Banco 3 | Banco 4 | Banco 5 [2] (pantalla) | Banco 6 | Banco 7 (pantalla) |
|---|---|---|---|---|---|---|---|---|
| 0xC000 | Banco 2 [1] | Estos 8 bancos pueden ser paginados sobre el area 0xC000-0xFFFF | | | | | | |
| 0x8000 | Banco 5 [2] (pantalla) | | | | | | | |
| 0x4000 | ROM 0 | ROM 1 | | | | | | |
| 0x0000 | | | | | | | | |

[1] -> El banco 2 que podemos mapear entre 0xC000 y 0xFFFF se corresponde con la misma zona de memoria 0x8000-0xC000.

[2] -> El banco 5 (Pantalla/VRAM) que puede ser mapeado en 0xC000 se corresponde con el mismo area de 0x4000-0x80000.

<---- La pagina entre 0x0000 y 0x4000 puede ser intercambiada entre 2 "bloques" o "paginas" que constituyen las 2 ROMs disponibles en estos modelos.

*Posibilidades de paginación*

**Cambiando de banco**

El puerto que controla la paginación en los modelos 128K es el 0x7FFD. La lectura del puerto no devolverá ningún valor útil, pero sí podemos escribir en él un valor con cierto formato:

| Bits | Significado |
|---|---|
| 0-2 | Página de la RAM (0-7) a mapear en el bloque 0xc000 - 0xffff. |
| 3 | Visualizar la pantalla gráfica "normal" (0) o shadow (1).<br>La pantalla normal está en el banco 5, y la shadow en el 7.<br>Aunque cambiemos a la visualización pantalla shadow, la pantalla<br>"normal" seguirámapeada entre 0×4000 y 0x7fff. |
| 4 | Selección de la ROM, entre (0) 128K menu, y (1) BASIC 48K. |
| 5 | Si lo activamos, se desactivará el paginado de memoria hasta<br>que se resetee el Spectrum. El hardware ignorará<br>toda escritura al puerto 0x7FFD |

A la hora de cambiar de página de memoria hay que tener en cuenta lo siguiente:

- La pila (stack) debe de estar ubicada en una zona de memoria que no vaya a ser paginada (no puede estar dentro de la zona que va a cambiar).
- Las interrupciones deben de estar deshabilitadas.
- Si se va a ejecutar código con interrupciones (y no pueden estar deshabilitadas), entonces debemos actualizar la variable del sistema 0x5B5C (23388d) con el último valor enviado al puerto 0x7FFD.

Un ejemplo de cambio de banco en ASM:

```
LD      A,(0x5b5c)      ;Previous value of port
AND     0xf8            ;only change bits you need to
OR      4               ;Select bank 4
LD      BC,0x7ffd
DI
LD      (0x5b5c),A
OUT     (C),A
EI
```

Podemos crearnos una rutina lista para usar con este código, como la que sigue:

```
;-------------------------------------------------------------------------
; SetRAMBank: Establece un banco de memoria sobre 0xc000
; Entrada   : B = banco (0-7) a paginar entre 0xc000-0xffff
; Modifica  : A, B, C
;-------------------------------------------------------------------------
SetRAMBank:
   LD A,(0x5b5c)                  ; Valor anterior del puerto
   AND 0xf8                       ; Sólo cambiamos los bits necesarios
   OR B                           ; Elegir banco "B"
   LD BC,0x7ffd
   DI
   LD (0x5b5c),A
   OUT (C),A
   EI
   RET
```

Un detalle apuntado por la documentación de World Of Spectrum es que los bancos 1, 3, 5 y 7 son "contended memory", lo que quiere decir que se reduce ligeramente la velocidad de acceso a estos bancos con respecto a los otros bancos. Un apunte muy importante es que en el caso del +2A y +3, los bancos de contended-memory ya no son el 1, 3, 5 y 7, sino los bloques 4, 5, 6 y 7. Al final de este capítulo veremos con más detalle qué es la contended-memory y en qué puede afectar a nuestros programas.

**Particularidades +2A/+3**

En el caso del +2A y +3 hay que tener en cuenta una serie de detalles "extra" a lo visto anteriormente, y es que estos 2 modelos tienen un modo de paginación especial, aunque siguen siendo compatible con el sistema de paginación que hemos visto. Por eso estos detalles que veremos a continuación son opcionales, ya que podemos utilizar el modo de paginación de la misma forma que en el +2 y 128K (paginación normal):

- Los bancos de contended-memory son los bloques 4, 5, 6 y 7 (no el 1, 3, 5 y 7).
- +2A y +3 tienen 4 ROMS en lugar de 2, por lo que el bit 4 del puerto 0x7FFD se convierte ahora en el bit bajo de la ROM a seleccionar, mientras que el bit alto se toma del bit 2.
- +2A y +3 tienen funcionalidades extra de paginado, que se controlan con el puerto 0x1FFD.

Este puerto (el 0x1FFD) tiene el siguiente significado a la hora de escribir en él:

| Bits | Significado |
|---|---|
| 0 | Modo de paginado (0=normal, 1=especial). |
| 1 | Ignorado en el modo normal, usando en el modo especial. |
| 2 | En modo normal, bit alto de la selección de ROM.<br>Usado de forma diferente en el modo especial. |
| 3-4 | 3 Motor del disco (1/0, ON/OFF), 4=Impresora |

Cuando se activa el modo especial, el mapa de memoria cambia a una de estas configuraciones, según los valores de los bits 1 y 2 del puerto 0x1FFD:

| | Bit 2 = 0<br>Bit 1 = 0 | Bit 2 = 0<br>Bit 1 = 1 | Bit 2 = 1<br>Bit 1 = 0 | Bit 2 = 1<br>Bit 1 = 1 |
|---|---|---|---|---|
| 0xFFFF | Banco 3 | Banco 7<br>(pantalla) | Banco 3 | Banco 3 |
| 0xC000 | Banco 2 | Banco 6 | Banco 6 | Banco 6 |
| 0x8000 | Banco 1 | Banco 5<br>(pantalla) | Banco 5<br>(pantalla) | Banco 7<br>(pantalla) |
| 0x4000 | Banco 0 | Banco 4 | Banco 4 | Banco 4 |
| 0x0000 | | | | |

*Posibilidades especiales de paginación*

Por otra parte, las 4 ROMS mapeables del +2A y +3 son:

| ROM | Contenido |
|---|---|
| 0 | Editor 128K, Menú y programa de testeo. |
| 1 | Chequeador de sintaxis 128K BASIC. |
| 2 | +3DOS. |
| 3 | BASIC 48K. |

**Ejemplo sencillo: alternando Bancos 0 y 1**

El siguiente ejemplo muestra la paginación de la siguiente forma:

- Paginamos el bloque/banco 0 sobre el área 0xC000-0xFFFF.
- Escribimos en memoria, en la posición 0xC000, el valor 0xAA.
- Paginamos el bloque/banco 1 sobre el área 0xC000-0xFFFF.
- Escribimos en memoria, en la posición 0xC000, el valor 0×01.
- Volvemos a paginar el banco 0 sobre el área de paginación.
- Leemos el valor de la posición de memoria 0xC000 y rellenamos toda la pantalla con dicho valor.
- Volvemos a paginar el banco 1 sobre el área de paginación.
- Leemos el valor de la posición de memoria 0xC000 y rellenamos toda la pantalla con dicho valor.

Haciendo esto, guardamos 2 valores diferentes en 2 bancos diferentes, y posteriormente recuperamos dichos bancos para verificar que, efectivamente, los valores siguen en las posiciones (0000) de los bancos y que la paginación de una banco a otro funciona adecuadamente. Se han elegido los valores 0xAA y 0×01 porque se muestra en pantalla como 2 tramas de pixeles bastante diferenciadas, siendo la primera un entramado de barras verticales separadas por 1 pixel, y la segunda separados por 7 pixeles.

Para terminar de comprender el ejemplo, lo mejor es ensamblarlo y ejecutarlo:

```
;-----------------------------------------------------------------------
; Bancos.asm
;
; Demostracion del uso de bancos / paginación en modo 128K
;-----------------------------------------------------------------------

 ORG 32000

 LD HL, 0
 ADD HL, SP                     ; Guardamos el valor actual de SP
```

```
  EX DE, HL                      ; lo almacenamos en DE

  LD SP, 24000                   ; Pila fuera de 0xc000-0xffff

  CALL Wait_For_Keys_Released
  LD HL, 0xc000                  ; Nuestro puntero

  ; Ahora paginamos el banco 0 sobre 0xc000 y guardamos un valor
  ; en el primer byte de sus 16K (en la direccion 0xc000):
  LD B, 0
  CALL SetRAMBank                ; Banco 0

  LD A, 0xAA
  LD (HL), A                     ; (0xc000) = 0xAA

  ; Ahora paginamos el banco 1 sobre 0xc000 y guardamos un valor
  ; en el primer byte de sus 16K (en la direccion 0xc000):
  LD B, 1
  CALL SetRAMBank                ; Banco 1

  LD A, 0x01
  LD (HL), A                     ; (0xC000) = 0x01

  ; Esperamos una pulsación de teclas antes de empezar:
  CALL Wait_For_Keys_Pressed
  CALL Wait_For_Keys_Released

  ; Ahora vamos a cambiar de nuevo al banco 0, leemos el valor que
  ; hay en 0xc000 y lo representamos en pantalla. Recordemos que
  ; acabamos de escribir 0x01 (00000001) antes de cambiar de banco,
  ; y que en su momento pusimos 0xAA (unos y ceros alternados):
  LD B, 0
  CALL SetRAMBank                ; Banco 0
  LD A, (HL)                     ; Leemos (0xc000)
  CALL ClearScreen               ; Lo pintamos en pantalla

  ; Esperamos una pulsación de teclas:
  CALL Wait_For_Keys_Pressed
  CALL Wait_For_Keys_Released

  ; Ahora vamos a cambiar de nuevo al banco 1, leemos el valor que
  ; hay en 0xc000 y lo representamos en pantalla. Recordemos que
  ; acabamos de leer 0xA antes de cambiar de banco, y que en su
  ; momento pusimos 0x01:
  LD B, 1
  CALL SetRAMBank                ; Banco 0
  LD A, (HL)                     ; Leemos (0xc000)
  CALL ClearScreen               ; Lo pintamos en pantalla

  ; Esperamos una pulsación de teclas:
  CALL Wait_For_Keys_Pressed
  CALL Wait_For_Keys_Released

  EX DE, HL                      ; Recuperamos SP para poder volver
  LD SP, HL                      ; a BASIC sin errores
  RET


;-----------------------------------------------------------------------
; SetRAMBank: Establece un banco de memoria sobre 0xc000
; Entrada: B = banco (0-7) a paginar entre 0xc000-0xffff
;-----------------------------------------------------------------------
SetRAMBank:
   LD A,(0x5b5c)                 ; Valor anterior del puerto
   AND 0xf8                      ; Sólo cambiamos los bits necesarios
   OR B                          ; Elegir banco "B"
   LD BC,0x7ffd
   DI
   LD (0x5b5c),A
   OUT (C),A
```

```
   EI
   RET


;-----------------------------------------------------------------------
; ClearScreen: Limpia toda la pantalla con un patrón gráfico dado.
; Entrada: A = valor a "imprimir" en pantalla.
;-----------------------------------------------------------------------
ClearScreen:
   PUSH HL
   PUSH DE
   LD HL, 16384
   LD (HL), A
   LD DE, 16385
   LD BC, 6143
   LDIR
   POP DE
   POP HL
   RET


;-----------------------------------------------------------------------
; Rutinas para esperar la pulsación y liberación de todas las teclas:
;-----------------------------------------------------------------------
Wait_For_Keys_Pressed:
   XOR A                       ; A = 0
   IN A, (254)
   OR 224
   INC A
   JR Z, Wait_For_Keys_Pressed
   RET

Wait_For_Keys_Released:
   XOR A
   IN A, (254)
   OR 224
   INC A
   JR NZ, Wait_For_Keys_Released
   RET

END 32000
```

El programa anterior, una vez ensamblado y ejecutado, esperará la pulsación de una tecla para mostrarnos en pantalla el valor de la celdilla de memoria 0xc000 mapeando primero uno de los bancos, y luego el otro.

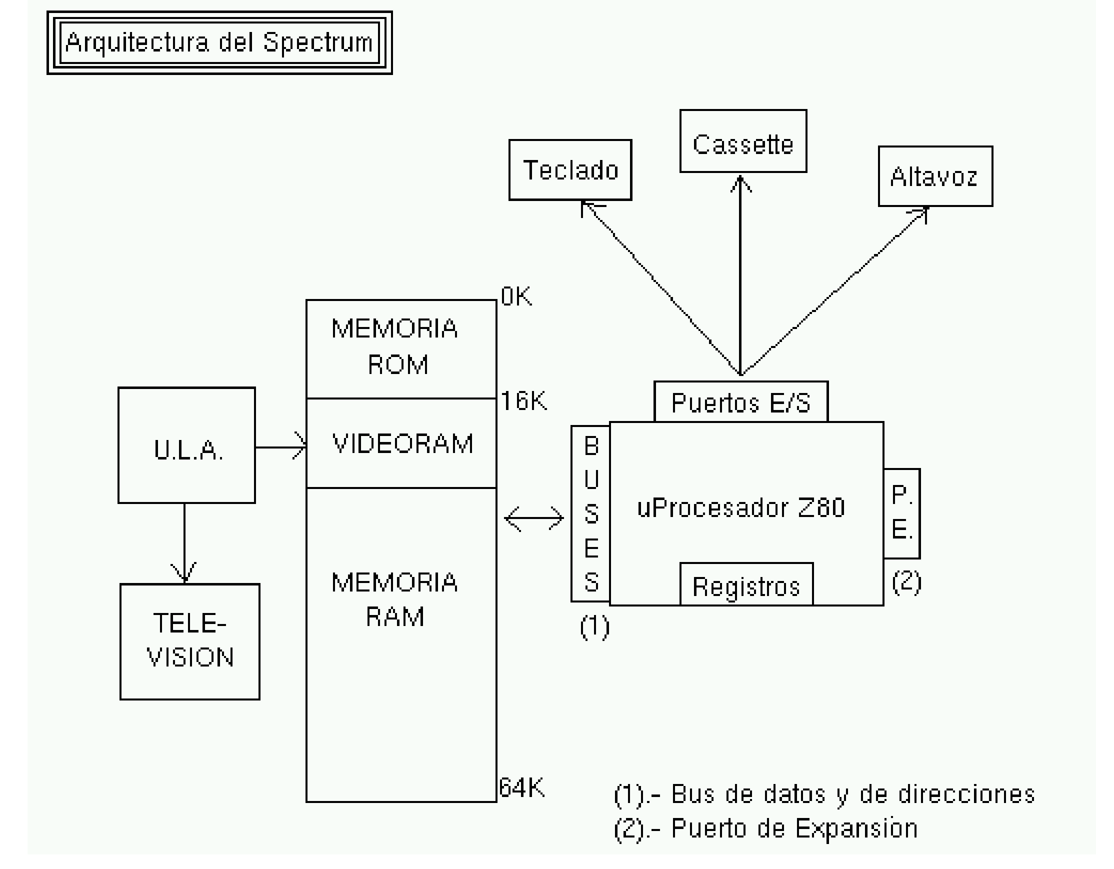
**Contended Memory**

En este artículo hemos hablado de la Contended Memory (podríamos traducirlo por "contención de memoria" o "memoria contenida").

Esta peculiaridad de la memoria del Spectrum puede traernos de cabeza en circunstancias muy concretas, como la programación de emuladores, o la creación de rutinas críticas donde el timming sea muy importante o donde tengamos que sincronizarnos de una forma muy precisa con algún evento.

El **efecto** es el siguiente: algunas zonas de memoria, en determinadas circunstancias, son de acceso más lento que otras en cuanto a ejecución de código que afecten a ellas (leer de esas zonas, escribir en esas zonas, ejecutar código que está en esas zonas). La **causa** es sencilla: una misma celdilla de memoria no puede ser accedida por 2 dispositivos diferentes simultáneamente.

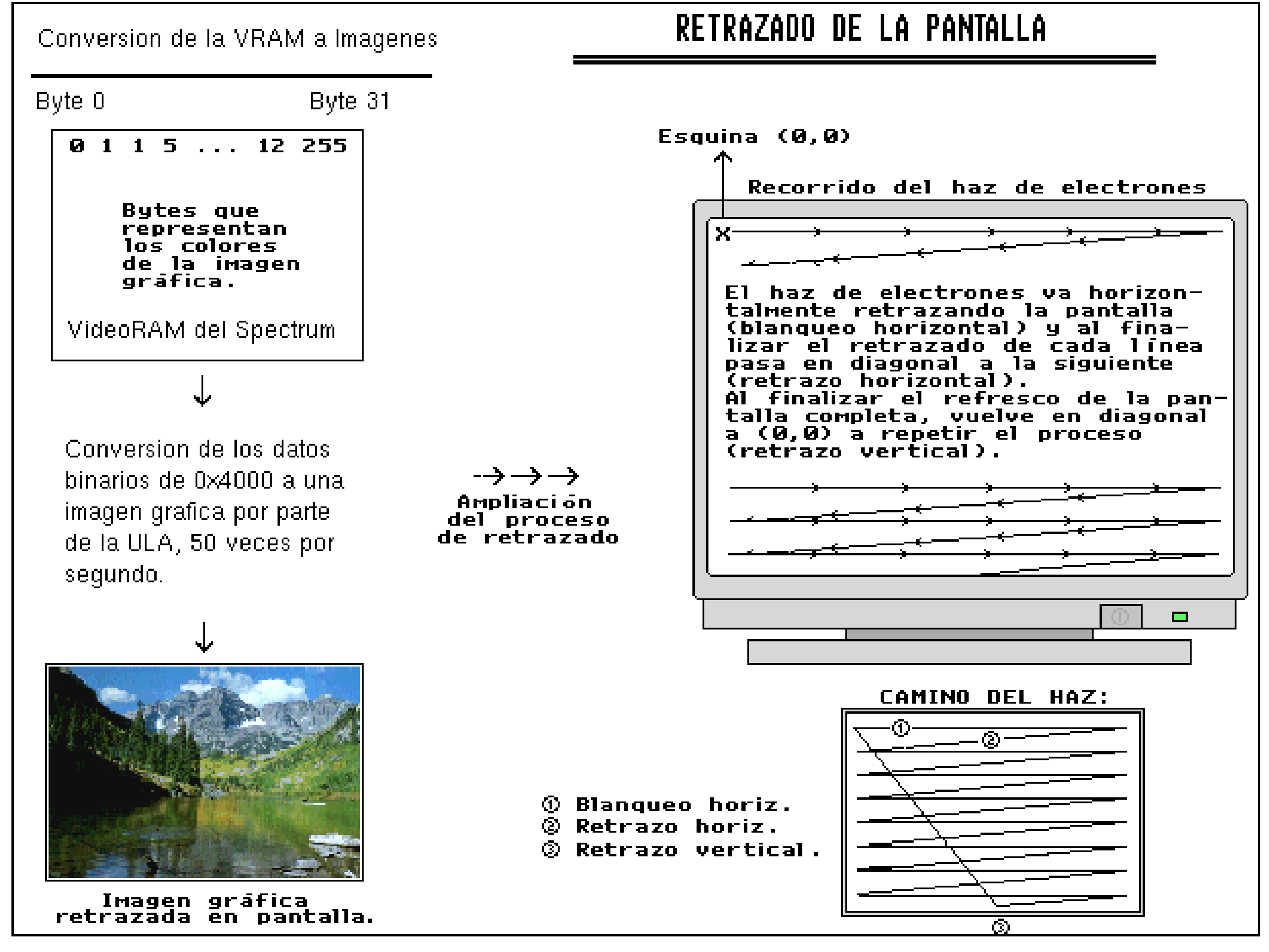
¿Acaso existen en el Spectrum otro dispositivo que acceda a la memoria además del microprocesador (cuando lee, decodifica y ejecuta instrucciones, o cuando lee/escribe en memoria)? Sí, lo hay, y es la ULA (Uncommited Logic Array).




*Arquitectura del ZX Spectrum*

La ULA es, digamos, "el chip gráfico" del Spectrum. Su labor no es como en los chips gráficos actuales o los chips gráficos de otros ordenadores (y consolas) de 8 bits, la de apoyar al software con funciones extra, sino que en el Spectrum la ULA se limita a leer la VIDEORAM (parte de la memoria que contiene la información gráfica a representar en el monitor), interpretarla, y mandar al modulador de TV las señales adecuadas para la visualización de dicha información en la TV. Sencillamente, es el chip que lee la VideoMemoria y la convierte en los píxeles que vemos en la TV.

¿Cómo trabaja la ULA? Este pequeño chip fabricado por Ferranti recorre 50 veces por segundo la zona de memoria que comienza en 0×4000 (16384) y transforma los datos numéricos en píxeles apagados o encendidos con el color de la tinta y papel asociado a cada celdilla 8×1 (8×8 en realidad) que va leyendo byte a byte, horizontalmente. Esto implica una sincronización con el haz de electrones del monitor de TV, que empieza en la esquina superior-izquierda y avanza horizontalmente hasta llegar al final de cada línea para, como en una máquina de escribir, pasar a la siguiente línea horizontal, y así hasta llegar hasta la esquina inferior derecha.




*Retrazado de la pantalla*

El problema es que mientras el haz de electrones del monitor avanza redibujando la imagen, el Spectrum no puede interrumpirlo (de hecho, no puede controlarlo, sólo se sincroniza con él) y tiene que servirle todos los datos necesarios para el retrazado de la imagen.

Esto implica que cuando la ULA está "redibujando" la pantalla (y recordemos que lo hace 50 veces por segundo) y por tanto leyendo de las sucesivas posiciones de memoria comenzando en 0×4000, el procesador no puede acceder a las celdillas exactas de memoria a las que accede la ULA hasta que ésta deja de hacer uso de ellas. En otras palabras, a la hora de leer celdillas de memoria entre 0×4000 y 0x7FFF, la ULA tiene prioridad sobre el procesador.

Por eso, los programas que corren en la zona de memoria entre 0×4000 y 0x7FFF pueden ser ralentizados cuando la ULA está leyendo la pantalla.

Como se detalla en la FAQ de comp.sys.sinclair alojada en World Of Spectrum, este efecto sólo se dá cuando se está dibujando la pantalla propiamente dicha, ya que para el trazado del borde la ULA proporciona al haz de electrones el color a dibujar y no se accede a memoria, por lo que no se produce este retraso o delay.

Controlar exactamente los retrasos que se producen y cómo afectarán a la ejecución de nuestros programas es un ejercicio bastante complejo que requiere conocimiento de los tiempos de ejecución de las diferentes instrucciones, número de ciclo desde que comenzó el retrazado de la pantalla, etc. Por ejemplo, una misma instrucción, NOP, que requiere 4 ciclos de reloj para ejecutarse en condiciones normales, puede ver aumentado su tiempo de ejecución a 10 ciclos (4 de ejecución y 6 de delay) si el contador de programa (PC) está dentro de una zona de memoria contenida, y dicho delay afectaría sólo al primer ciclo (lectura de la instrucción por parte del procesador). Por contra, si en lugar de una instrucción NOP tenemos una instrucción LD que acceda a memoria (también contenida), el delay puede ser mayor.

Como podéis imaginar, este es uno de los mayores quebraderos de cabeza para los programadores de emuladores, y es la principal causa (junto con la **Contended I/O**, su equivalente en cuanto a acceso de puertos, también producido por la ULA), de que hasta ahora no todos los juegos fueran correctamente emulados con respecto a su funcionamiento en un Spectrum. También muchas demos con complejas sincronizaciones y timmings dejaban de funcionar en emuladores de Spectrum que no implementaban estos "retrasos" y que, en su emulación "perfecta del micro", ejecutaban siempre todas las instrucciones a su velocidad "teórica". En nuestro caso, como programadores, la mejor manera de evitar problemas en la ejecución de nuestros programas es la tratar de no situar código, siempre que sea posible, entre 0×4000 y 0x7FFF.

**Contended Memory + Paginación**

¿Cómo afecta la contended-memory al sistema de paginación de los modelos 128K? Al igual que en el 48K

existe una ¿página¿ (0×4000-0x7FFF) a la que la ULA accede y por tanto afectada por sus lecturas, en el caso de los 128K existen bancos de memoria completos (páginas) de memoria contenida. Como ya hemos visto, estos bancos son:

- Modelos +2/128K : Bancos 1, 3, 5 y 7.
- Modelos +2A/+3: Bancos 4, 5, 6 y 7.

La afectación de velocidad de lectura de esta memoria es más importante de lo que parece. Según el manual del +3:

```
The RAM banks are of two types: RAM pages 4 to 7 which are contended
(meaning that they share time with the video circuitry), and RAM pages
0 to 3 which are uncontended (where the processor has exclusive use).
Any machine code which has critical timing loops (such as music or
communications programs) should keep all such routines in uncontended
banks.

For example, executing NOPs in contended RAM will give an effective
clock frequency of 2.66Mhz as opposed to the normal 3.55MHz in
uncontended RAM.

This is a reduction in speed of about 25%.
```

Es decir, la velocidad de acceso a memoria (y por tanto, también de ejecución) cae un 25% en un banco con contended-memory con respecto a un banco que no lo sea.

El problema, que para el 128K y +2 las páginas que sufre una penalización son unas, y para el +2A y +3 otras diferentes, por lo que parece que siempre tendremos que primar a uno de los modelos sobre otros: o usamos números de bancos que no penalicen al +2A/+3, o lo hacemos para el 128K/+2.

Lo mejor es no situar en la zona paginada rutinas como las de vídeo o audio, o al menos, no hacerlo si éstas son críticas. En cualquier caso, es probable que para el 90% de las rutinas o datos de un programa no existan problemas derivados de correr o estar alojados en memoria contenida, pero puede ser un detalle a tener muy en cuenta en rutinas que requieran un timming perfecto (efectos con el borde, efectos gráficos complejos, etc).

**Paginación de memoria desde Z88DK (C)**

Podemos paginar memoria también desde C usando Z88DK mediante un código como el siguiente:

```
//--- SetRAMBank ------------------------------------------------------
//
// Se mapea el banco (0-7) indicado sobre 0xC000.
//
// Ojo: aqui no se deshabilitan las interrupciones y ademas en lugar
// de usar el registro B, se usa un parametro tomado desde la pila.
// En caso de ser importante la velocidad, se puede usar "B" y no pasar
// el parametro en la pila, llamando SetRAMBank con un CALL.
//
void SetRAMBank( char banco )
{
  #asm
   .SetRAMBank
     ld hl, 2
     add hl, sp
     ld a, (hl)

     ld b, a
     ld  A, ($5B5C)
     and F8h
     or  B
     ld  BC, $7FFD
     ld  ($5B5C), A
     out (C), A
   #endasm
}
```

Con el anterior código podemos mapear uno de los bancos de memoria de 16KB sobre la página que va desde 0xC000 a 0xFFFF, pero debido al uso de memoria, variables y estructuras internas que hace Z88DK, debemos seguir una serie de consideraciones.

- Todo el código en ejecución debe estar por debajo de 0xC000, para lo cual es recomendable definir los gráficos al final del ¿binario¿.
- Es importantísimo colocar la pila en la memoria baja, mediante la siguiente instrucción (o similar, según la dirección en que queremos colocarla) al principio de nuestro programa:

```
/* Allocate space for the stack */
#pragma output STACKPTR=24500
```

La regla general es asegurarse de que no haya nada importante (para la ejecución de nuestro programa) en el bloque 0xC000 a 0xFFFF cuando se haga el cambio: ni la pila, ni código al que debamos acceder. Tan sólo datos que puedan ser intercambiandos de un banco a otro sin riesgo para la ejecución del mismo (por ejemplo, los datos de un nivel de juego en el que ya no estamos).

**En resumen**

Comprendiendo el sistema de paginación de los modelos de 128K y aprendiendo a utilizarlo conseguimos una gran cantidad de memoria adicional que ir paginando sobre el bloque 0xC000-0xFFFF.

Así, podemos almacenar los datos de diferentes niveles en diferentes bloques, y cambiar de uno a otro mediante paginación en el momento adecuado. Esto permite realizar cargas de datos desde cinta almacenando la totalidad de los datos del juego o programa en bancos libres de memoria y convertir nuestro juego multicarga (con una carga por fase) en un juego de carga única (con todos los elementos del juego almacenados en memoria), evitando el tedioso sistema de rebobinar y volver a cargar la primera fase cuando el jugador muere.

Ahora bastará con que nuestro programa, una vez cargado en memoria y en ejecución, pagine un determinado bloque, cargue 16K-datos sobre él, pagine otro bloque diferente, y realice otra carga de datos desde cinta, y así sucesivamente con todos los bloques de datos del juego. Estas cargas de datos podemos hacerlas bien desde nuestro programa ¿principal¿ una vez cargado y en memoria, o bien desde un mini-programa lanzado por el cargador BASIC y previo a cargar el programa definitivo.

El resultado: 128KB de memoria a nuestro alcance, tanto para cargar múltiples datos gráficos o de mapeado sobre ellos como para llenarlos internamente desde nuestro programa.

**FICHEROS**

- Ejemplo de paginación / cambio de bancos.
- Fichero tap del ejemplo anterior.

**LINKS**

- FAQ 128K de WOS.
- FAQ 48K de WOS.
- Contented Memory,
- Manual del +3.
- Puertos E/S.
- Web del Z80.
- MH 137: La memoria paginada del Spectrum 128K (página 1).
- Z80 Reference de WOS.

Santiago Romero







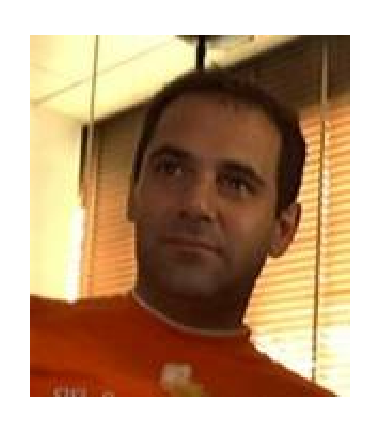


# Entrevista a Ignacio y Carlos Abril

*Los inicios en el mundo de la informática de los hermanos Abril no fueron muy diferentes de los de otros chavales de la época. Pero su inquietud en querer desarrollar videojuegos y su afortunada decisión de llamar a la puerta de Dinamic Software para mostrarles sus trabajos fueron suficientes hechos diferenciadores para formar parte de la historia del software español. De todo esto y mucho más nos hablan en la siguiente entrevista que amablemente nos han concedido.*

«Ahora uno se pregunta cómo era posible meter en 48K todo lo que se metía.»





«Eramos chavales y no había planteamiento estratégico pero lo hicimos muy bien.»

**Como muchos desarrolladores de la época, empezaríais programando en vuestra casa. ¿Cómo conseguisteis vuestro primer ordenador?**

*Carlos*: Nuestro primer ordenador lo conseguimos porque, después de mucho pedirlo, nuestros padres nos lo prometieron si al final de curso sacábamos buenas notas. Así que nosotros cumplimos con creces y mis padres, a pesar de que no pasábamos una buena situación económica en la familia, hicieron un esfuerzo y cumplieron también. Un ZX Spectrum 48K que enchufábamos a la tele que compartíamos con el resto de un total de 9 hermanos (aunque, afortunadamente, en todos los sentidos, en mi casa hemos sido de los que casi no veíamos la tele).

*Ignacio*: El precio fue en torno a las 42.000 pesetas, con gran susto por parte de nuestra madre que le parecía un disparate. Afortunadamente nuestro padre trabajaba con ordenadores y sabía que eso era el futuro.

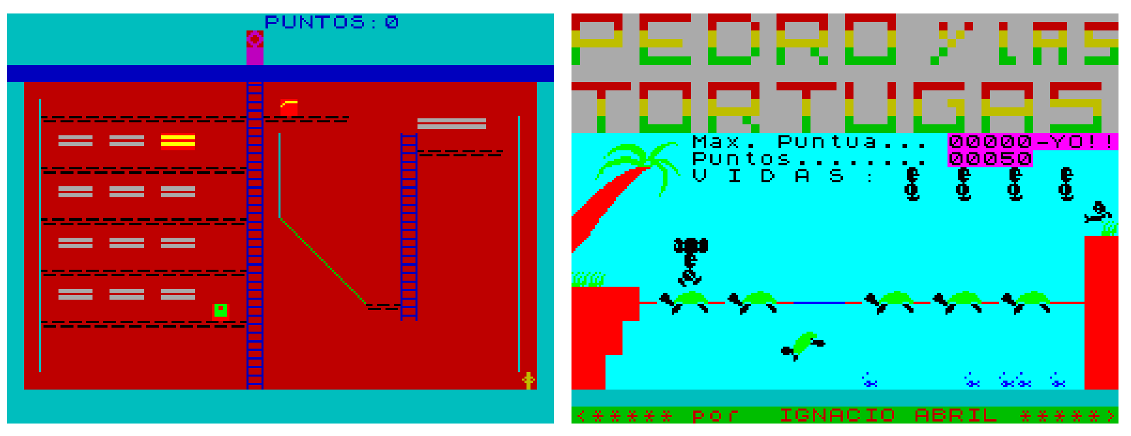
**Vuestros inicios (Ladrón y Pedro y las tortugas) fueron publicados en 2 importantes revistas del sector: Microhobby y Micromanía. Al verlos publicados, ¿ya preveíais la posibilidad de dedicaros profesionalmente a la programación?**

*C*: En concreto mi programa: 'Ladrón' estaba hecho en Basic para el Spectrum cuando todavía no había visto ningún juego para ese ordenador. El ordenador lo tuvimos a principio de verano, cuando acabó el curso, y los primeros juegos comerciales no los vimos hasta el final del verano.

Así que después de haberme estado dejando la vida en Basic para conseguir mover dos caracteres independientes por la pantalla e iluminar unas ventanitas, cuando vi los primeros juegos comerciales, no comprendía nada. Tenía unos 14 años. Fue mi hermano el que me dijo que creía que había una cosa que se

llamaba código máquina en la que se hacían los juegos y rápidamente nos pusimos a buscar información.

*I*: No nos planteábamos el que fuera nuestro futuro profesional, sino, simplemente un hobby. Nos gustaba programar y ver el resultado de tu esfuerzo. El hecho de que salieran en las revistas más importantes de aquel entonces era algo que nos emocionaba. A la vez, nos dio la posibilidad de comprarnos una pequeña televisión de 14" que nos daba una cierta libertad para no tener que estar en medio del salón de casa programando.




*Ladrón (de Carlos Abril) y Pedro y las tortugas (de Ignacio Abril) sentaron las bases*

**Nonamed fue la carta de presentación de Ignacio con Dinamic. ¿Cómo se produjo ese primer contacto? ¿Se tuvieron que hacer muchos cambios en el juego o aceptaron la versión presentada? ¿Lo desarrollaste íntegramente de modo casero o recibiste ayuda de Dinamic?**

*C*: Nonamed se hizo inicialmente como un jueguecillo sin grandes aspiraciones. Estaba hecho con el objetivo de ver si podía sacarse bajo un sello que había sacado Dinamic en aquel entonces que se llamaba 'Future Stars'. Cuando lo llevamos y luego nos llamaron nos dijeron que 'Future Stars' había sido un fracaso pero que el juego tenía potencial para sacarlo bajo Dinamic aunque no servían los gráficos y el desarrollo era demasiado simple. Los gráficos los rehicieron todos en Dinamic y eso ayudó a cambiar el desarrollo.

En aquel entonces hacíamos los juegos en casa, con ayuda gráfica si era necesaria. Incluso posteriormente había programadores a los que se les facilitaban algunas rutinas para ayudarles. Pero los juegos los hacíamos con nuestros propios medios y luego cobrábamos, por royalties, cuando empezaban a venderse.

*I*: Dinamic, por aquel entonces, estaba en la Torre de Madrid, en la planta 31 (si no recuerdo mal). Fuimos allí Carlos, un amigo y yo. Recuerdo que estábamos emocionados por poder acceder al "templo" de Dinamic. Cuando entramos nos recibió Nacho Ruiz que fue quien nos cogió el juego. Fuimos hacia el final de la sala y allí vimos a Victor mapeando los enemigos en el Army Moves, concretamente en la fase del jeep en la que salen aviones.

Sobre el Nonamed, los gráficos iniciales eran los que Dinamic hizo para la revista Microhobby y que salieron en el número 76 (http://www.microhobby.org/numero076.htm, en la página 22 y siguientes). Como dice Carlos, luego fueron cambiados completamente.

El sistema de desarrollo que usábamos era un Spectrum con una cinta de casete. Eso suponía que había que cargar el editor desde la cinta, el programa fuente, modificarlo y grabar los cambios; compilarlo; grabar el ejecutable obtenido; cargar los gráficos, mapa y demás: ejecutar el programa y, cuando petaba o terminabas la sesión de depuración, volver a comenzar. Vamos, un infierno en toda regla, aunque por aquel entonces nos parecía lo normal. Eso sí, tenías que tener una buena "gestión de cintas de casete", para que no se te fuera al garete todo el trabajo.

**Apenas un año después de Nonamed, Phantis, realizado por Carlos, fue lanzado comercialmente. Este juego, ¿ya fue creado dentro de Dinamic o siguió un proceso de desarrollo similar a Nonamed?**

*C*: Cuando fuimos con Nonamed a Dinamic, yo ya tenía Phantis bastante avanzado. Eran dos planteamientos diferentes. El primero era un juego sencillo para la firma 'Future Stars', para ver un poco como funcionaba el tema y ver si se podían vender juegos. Y el segundo era un planteamiento de juego bueno para la época.

La diferencia fundamental entre mi hermano y yo era que a mí, además de programar, me encantaba hacer gráficos y eran bastante buenos, algo que diferenciaba el producto. Yo podía querer un gráfico aquí o un enemigo allá o cambiar una cosa o no cambiar una animación que no quedaba bien y lo hacía y lo metía en el juego. Nacho era más dependiente en ese aspecto. La primera vez que llevé y enseñé el juego a Dinamic, ya estaba prácticamente terminado.

Después de esa presentación, el juego les encantó y yo cogí y lo cambié todo: distribución del mapa, había un vehículo terrestre espacial en la gruta que quité, había unas medusas bastante grandes que quité, puse una chica en vez del astronauta que era el personaje principal, sólo para que Azpiri me hiciese una portada más 'espectacular', etc. ¿Por qué esos cambios que nadie me había pedido? Yo creo que es algo que nos sigue pasando: enseñas el juego y al enseñarlo y tener que explicarlo te das cuenta de algunas cosas que antes no se habían pensado.

**En Inglaterra, Phantis fue lanzado como Game Over II, adaptándose tanto gráficos, portada e instrucciones a esta nueva denominación. ¿Qué implicación tuviste en esta decisión?**

*C*: Phantis se vendió bien. Pero en aquel entonces había otro juego que internamente, en Dinamic, era considerado inferior y vendió un poco más. El motivo que se barajó en aquel entonces fue que los jugadores eran principalmente chicos que no se sentían identificados con un personaje femenino.

Además, me dijeron que no saben cómo no habían pensado antes en usar ese nombre que hubiese sido más notorio. A mí me gustaba Game Over, así que la idea me pareció excelente.

**¿Qué impresión os causó conocer y trabajar con los hermanos Ruiz?**

*C*: El día que fuimos por primera vez a Dinamic teníamos como 15 ó 16 años, yo, y uno más mi hermano Nacho. Nos abrió Nacho Ruiz quien básicamente cogió el juego, no me acuerdo si llegó a cargarlo y nos dijo que ya nos dirían algo. Víctor Ruiz estaba mapeando enemigos del Army Moves y al vernos, éramos tres personas, dijo algo así como: "¡Vaya! Una invasión", se levantó y desapareció.

Recuerdo que después de pasado el primer flash de haber conocido a los que habían hecho el Saimazoom y Babaliba y que acababan de sacar el Profanation, nos quedamos un poco decepcionados por el nivel técnico que tenían. Básicamente las rutinas complejas las hacía otra persona y me quedé alucinado cuando salió Profanation porque el movimiento de sprites y el parpadeo que hacían en la pantalla estaba muy por debajo de lo que nosotros ya hacíamos en aquel entonces.

Posteriormente con Víctor tuve una buena amistad: era una persona muy sociable, flexible, que escuchaba con paciencia y comprendía la diversidad psicológica de la gente. He estado trabajando con ellos desde entonces hasta principios de 2007 que fue cuando me fui de FX. Eso son casi 25 años haciendo cosas juntos, incluso fundar Dinamic Multimedia y FX Interactive, por lo que hemos tenido muy, muy buenos momentos, hemos creado grandes cosas y les deseo lo mejor en sus vidas.

El que más nos llamó la atención cuando empezamos a trabajar en Dinamic fue Jesús Alonso. Los hermanos Ruiz eran muy callados y cuando llegamos allí, después de aquel breve primer día, a ver el juego con ellos y mientras cargaba de la cinta del Spectrum, el silencio fue roto por una persona que se presentó como Jesús Alonso y que no paró de hablar las siguientes dos horas: emanando motivación e ilusión por todos lados, describiéndonos todo como increíble y alucinante y describiéndonos todo el proceso que ellos hacían con los juegos y sus planes presentes y futuros. Cuando salimos mi hermano y yo estuvimos todo el viaje de metro hablando de lo estupendo de la experiencia y de lo bien que nos caía aquel tipo. Recuerdo que dos años más tarde nos invitaron a la 'cena corporativa' de Dinamic y cuando nos hicieron presentarnos mi hermano agradeció a Jesús Alonso el habernos abierto las puertas y lo definió como la persona más carismática que había conocido.

Posteriormente he tenido el placer de tenerlo como socio en FX Interactive y a principios de febrero estuve en una charla donde explicaba su propuesta empresarial en 'http://www.restaurantes.com' y sigue inyectando la misma pasión e ilusión que hace veinticuatro años.







*Nonamed y Phantis fueron sus primeros juegos comerciales*

**Ignacio, ¿cómo se produjo la proposición de realizar el primer Navy Moves?**

*I*: La verdad es que fue muy simple. Una vez que había terminado con la conversión del Nonamed para Amstrad CPC, Victor me propuso hacer la segunda parte del Army Moves. La verdad es que yo acepté inmediatamente, porque el Army Moves me gustaba mucho y tener la posibilidad de participar en la programación de su continuación me pareció una oportunidad imposible de rechazar. Para mí fue muy excitante y cuando llegué a casa con algunos disquetes con gráficos del Amstrad CPC para el juego, llamé emocionado a Carlos para que viera lo que me habían propuesto.

**El cambio de desarrollar en vuestra casa con el ZX Spectrum a poder desarrollar con las herramientas facilitadas por Dinamic sería notorio. ¿Qué modo de trabajo había en Dinamic?**

*C*: Nosotros nunca hemos trabajado en Dinamic en la época de Dinamic. Siempre trabajamos en casa, sin ninguna vinculación laboral. Sí cambió mucho que nos ofrecieran comprar unos Amstrad CPC con disquete y aquello fue un cambio radical. Posteriormente compraron PCs (Amstrad 640) y nos ofrecieron comprarlos. Y también pusieron a nuestra disposición un sistema de programación (PDS - Programmer developer System) con el que escribías el código en PC, luego se ensamblaba en el propio PC, se mandaba por un cable paralelo y se ejecutaba en la máquina para la que estuvieses desarrollando: Amstrad CPC, Spectrum o MSX.Y posteriormente Amiga y Atari ST. Nosotros éramos programadores 'ajenos' a Dinamic y, por tanto, todo nos lo ofrecían por si queríamos comprarlo para utilizarlo.

Pero qué duda cabe que el ponerlo a nuestra disposición fue una ayuda inestimable. Lo de ensamblar en un PC, con disco duro, que se ejecutase en otra máquina que se terminaba reseteando (funcionase o no funcionase lo probado) y no tener que volver a cargar el ensamblador, el código fuente, etc. ¡No me imagino haber tenido eso al principio! En el Spectrum, escribíamos código, grabábamos en casete, ensamblábamos, ejecutábamos, reseteábamos, sacabas la cinta del código, metías la del ensamblador, la rebobinabas, la cargabas, luego metías la cinta del código, rebobinabas un poco a ojo (si rebobinabas mucho cargabas una versión anterior), luego la sacabas, metías la cinta con los gráficos, mapas, etc, cargabas todo eso en la posición que querías de memoria, cambiabas el código fuente, ensamblabas y volvías a empezar. Un infierno.

Nosotros éramos jóvenes, en la época del Phantis tenía unos 16 años, pero el director general de Dinamic tenía 19… Eso de modos de trabajo, procedimientos, metodologías, … No existía ni en Dinamic ni en ninguna otra compañía de desarrollos de juegos que conozca de aquel entonces, española o no.

Por 1995, unos ocho años más tarde, fue cuando empecé a interesarme por la producción, por ingeniería de software y si algo era aplicable al desarrollo de juegos y muchas más cosas. Y seguía prácticamente sin hablarse de eso en la incipiente industria del PC y de la primera Playstation. He ido a compañías de las de más renombre o incluso en el año 2000 en mi primer GDC y lo de la producción existía como título pero los procesos, la división metódica de los pipelines y de sus procedimientos y todas esas cosas, incluso cual era el papel del propio productor, estaban muy, muy verdes. Afortunadamente desde entonces la cosa ha mejorado bastante pero aún hablas con productores de juegos que no saben qué tienen que hacer o que se equivocan por falta de formación.

Los procesos de producción, el diseño previo detallado, la división de tareas, las estimaciones de tiempo, etc. las introduje yo en Dinamic Multimedia, por la época de PCFútbol 4 y 5.

Al principio, por ejemplo, cuando llegamos a Dinamic les pedimos ver la herramienta que tenían para hacer gráficos, para almacenarlos y cogerlos para usarlos en el juego. Resultó ser un programa comercial al que le habían añadido la opción de poder coger los gráficos. En cambio, la herramienta que yo había hecho era infinitamente mejor porque estaba hecha por y para nosotros y como la utilizábamos mucho la habíamos dejado cómoda y funcional. Los ensambladores eran los de HiSoft que eran los que usaba todo el mundo.

Cuando saqué Phantis me compré un Amiga 500, que acababa de salir en España, y esas Navidades estaban de promoción. Fui el primero de la gente que trabajaba con Dinamic y del propio Dinamic en hacer cosas para el Amiga e hice las herramientas para cargar los gráficos del Deluxe Paint, coger gráficos para el juego e incluso funcionar por el puerto serie. Se lo dejé a la gente de Dinamic y se lo pasaron a otra gente que trabajaba para ellos fuera. Igual con las rutinas de pintado que utilizaban el blitter, sonido, etc.

Que recuerde el único que hizo un uso más allá de las rutinas que yo había hecho, mejorando y trabajando con los sprites hardware de la máquina fue Pablo Ariza con el Astro Marine Corps. El resto las utilizaron tal cual.

**¿Hubieron muchos cambios a lo largo de su proceso de desarrollo? ¿Tenías cierta libertad de decisión o ya había unas directrices marcadas?**

*C*: Nonamed cambió, como he dicho, el planteamiento y acabó siendo más de Dinamic, pero ese era el objetivo. Eramos chavales y no había planteamiento estratégico :-) pero viéndolo desde la distancia lo hicimos muy bien: un producto sencillo y no arriesgado para conocer a la gente, ver cómo funciona, si es un buen negocio y un producto más complejo para después, cuando ya estábamos establecidos. Phantis cambió porque quise pero no me impusieron ni pidieron ningún cambio relevante. Eran nuestros juegos y nos dejaban hacer lo que creíamos.

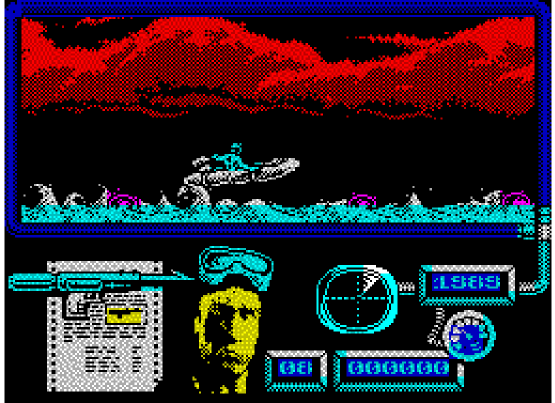
*I*: En el caso del Navy Moves, Victor era el productor y diseñador del juego. Yo me limitaba a programar aquello que él diseñaba, aunque, indudablemente, metía cosas de mi cosecha en cuanto al diseño del mapa o mapeado y ataque de enemigos, mandos superiores… Era un poco como en el caso de los grafistas, Victor les explicaba la idea que quería y ellos creaban los gráficos con su sello característico.

**Durante el desarrollo seguro que surgieron innumerables problemas o limitaciones debido a la propia máquina. ¿Hubo alguno en concreto que diera cierta guerra?**

*C*: Problemas como tal, no, pero limitaciones todas. ¡Eran máquinas de 48KB! Lo bueno del Spectrum, el Amstrad, el Amiga 500 o el Atari ST es que todas eran iguales entre sí. Todos los Spectrum eran iguales, todos los Amstrad iguales, Amiga, Atari ST, igual: la misma memoria, la misma velocidad de CPU, la misma resolución de pantalla, etc. No es como ahora con el PC que tienes que tener en cuenta al que tenga una máquina lenta, con tarjeta gráfica lenta, etc.

Pero claro, hablamos de una memoria de 48KB en total (incluyendo la pantalla) para las máquinas de 8 bits. Pues había que estar aprovechando cada byte. Y si veías una animación y querías meter un frame más pues posiblemente no podías. Se trabajaba para reducir el número de frames en una animación y que quedara lo mejor posible. Y la velocidad de la CPU…

*I*: La verdad es que visto ahora en la distancia uno se pregunta cómo era posible meter en 48k todo lo que se metía y cómo era posible que con esas CPUs los gráficos se movieran relativamente suaves.

*Navy Moves: un juego emblemático de Dinamic y de la historia del software español*

**Por fin, en 1988, se publica el juego en una presentación de lujo en caja de cartón grande con cantidad de extras y documentos, además de regalar el juego Army Moves. Dinamic no escatimó en recursos para hacer que Navy Moves fuera uno de los grandes…**

*C*: El único problema real, por el cual todavía incluso nos reímos y que recuerdo que recordábamos en una reunión de FX hace unos tres años fue una frase que ponía en la caja y que decía: "No podrás soportarlo…" ¡A quién se le ocurrió aquella frase y tuvo la osadía de ponerla allí!

*I*: La verdad es que el producto quedó muy bien. Eso es algo que tanto Dinamic en su tiempo como FX Interactive han sabido cuidar mucho: el ofrecer a los usuarios un producto de lujo y un servicio técnico impecable.

**Una queja común respecto a los juegos españoles es la dificultad de ciertos juegos. En el caso de Navy Moves, es cierto que puede resultar desesperante al principio. ¿Teníais algún tipo de baremo para establecer el nivel de dificultad?**

*C*: Esa fue una de mis preguntas a Nacho Ruiz cuando establecimos amistad: ¿de verdad os acababais juegos como Game Over, Camelot Warriors, …? Y su respuesta fue clara. Me dijo que no, que hacían los juegos imposibles porque parecía que el reto es lo que le gustaba a la gente y que aún así había gente que, no se explicaban cómo, pero se los acababan. Ellos se lo acababan cuando lo desarrollaban con el típico 'poke' de invencibilidad o similar. Phantis fue el primer juego de Dinamic que rompió radicalmente con este 'concepto'.

*I*: La verdad es que cuando uno juega cien mil veces a una fase, al final se la pasa. Tal es el caso de la primera fase de la zodiac en Navy Moves. Al mapear las minas reconozco que no me pareció especialmente difícil, porque eran muchas horas en las que me dedicaba a testear la fase y ya tenía todo medido. Sin embargo, también es cierto que, con motivo de su salida en Inglaterra, fuimos a Londres a enseñarlo a las revistas y al jugar yo a la fase de las minas me fue imposible pasarla. Un bochorno.

**Vuestro esfuerzo se ve recompensado al ser Navy Moves premiado con, entre otros premios, el joystick de oro al mejor programa del año 88 por la revista Microhobby. ¿Qué implicó recibir aquel premio?**

*C*: El premio fue fundamentalmente para mi hermano. Yo sólo hice las versiones de Amiga y de MSX. En la versión de Amiga, en la segunda carga, cambié la mayoría de los gráficos porque, como gran fan del Amiga, me oponía a que los gráficos fuesen la versión de los de Spectrum coloreados con 16 colores que hacían para Atari ST y Amiga. El Amiga tenía modos de 32 y 64 colores y de una paleta mucho mayor. Yo los convertí a 32 colores y retoqué muchos gradientes, 'antialiasings', etc.

Pero recoger aquel premio fue una fiesta muy divertida con gente de Hobbypress que todavía hoy continúa y conocer a otras personas de otras compañías.

*I*: Fue un reconocimiento a un trabajo divertido, aunque duro. Por aquel entonces yo estaba estudiando a la vez que hacía el juego, por lo que el esfuerzo realizado fue bastante importante. Como dice Carlos, la fiesta fue una oportunidad de conocer a otras personas dentro de la incipiente industria del desarrollo de videojuegos en España.

**Una curiosidad: los famosos nombres FX Synchro Sprites, FX Doble Carga o FX E.A.G. (Enemigos Auténticamente Gigantescos) que adornaban las carátulas de varios juegos de Dinamic, ¿es posible saber de dónde surgen?**

*C*: Esos eran nombres que la gente de Dinamic se inventaba. En general se ponían nombre graciosos a las cosas y luego, si quedaban bien traducidos o como acrónimos se usaban, si resultaban muy soeces pues se les daba la vuelta o similar. Por ejemplo, cuando desarrollábamos Phantis, al 'caballito' que salía al final de la primera carga, le llamábamos 'la cerda'. Así que en las instrucciones es el Adrec Cónico que si os fijáis es cerda escrito al revés.






*PC Fútbol: Una franquicia de Dinamic Multimedia en la que participaron Carlos e Ignacio*

**Hasta el ocaso de los ordenadores de los 8 bits y el paso a Dinamic Multimedia (y con ello a la realización mayoritariamente de simuladores deportivos), ¿en qué desempeños anduvisteis involucrados?**

*C*: Nacho y yo estuvimos trabajando en la empresa donde trabajaba mi padre haciendo la parte informática de un levantamiento topográfico. En casa estuve haciendo pruebas de programación para PC, 'scroles' en varios planos usando la VGA en 256 colores, y personajes que se movían. Y para crear las herramientas, usando la VGA en 16 colores, hice todo un sistema de 'user interface', con botones, listas, fuentes de texto (en aquel entonces no había Windows, era MS-DOS), gráficos, etc, que fue lo que Víctor Ruiz vio cuando vino a casa a pedirme la Megadrive y lo que dio lugar a que hiciésemos entre otras cosas, el catálogo de McGraw-Hill, que fue lo que desembocó en el Simulador Profesional de fútbol y en la posterior creación de Dinamic Multimedia y el desarrollo de PCFútbol 2.0.

*I*: Desde que dejé Dinamic, no volví a hacer nada referente a los juegos hasta 1999 cuando entré en FX Interactive. En ese tiempo estuve dedicado a desarrollar programas en el campo de la ingeniería civil. Uno de los proyectos más interesantes en los que estuve fue desarrollar el centro de control de un proyecto de seguimiento de coches dotados con GPS y sistema de comunicación, pionero en aquel momento.

**El salto de los ordenadores de 8 bits a ordenadores de 16 bits, ¿supuso un gran cambio en vuestra forma de desarrollar?**

*C*: Personalmente fue un gusto. El assembler del 68000 ha sido el que más me ha gustado con gran diferencia en toda mi vida. El paso al assembler del 8086 de los PC fue un paso atrás bastante 'desagradable'. El Amiga ha sido el ordenador con el que más he disfrutado desarrollando.

*I*: El hecho de tener mucha memoria disponible y un procesador con instrucciones mucho más complejas, nos abrió un campo de posibilidades a desarrollar.

**¿Cómo vivisteis la desaparición de Dinamic como empresa?**

*C*: En mi caso, no la sufrí en exceso. Yo había hecho Phantis y ya era un juego que no se vendía. Después estuve desarrollando la idea original y programando Lorna en Amiga y cuando dije que no me interesaba hacerlo (básicamente si mi royaltie en cualquier juego hubiese sido del 30%, al ser Lorna se reducía a la mitad, ya que la otra mitad era para Azpiri que en aquel entonces ya estaba metido con Mot y no podía dedicarle todo el tiempo que hubiese sido razonable) aproveché todo lo que había hecho de Amiga para hacer rápidamente en esa máquina, la versión de Navy Moves que estaba acabando mi hermano. Luego sacamos el juego pero la compañía estaba en crisis así que no conseguimos que nos pagaran mucho.

Pero yo lo sufrí menos porque, al menos, había hecho un juego que se había vendido bien y por el que me habían pagado, con lo cual las cosas que me había comprado (ordenadores, el Amiga, monitores, etc.), estaban pagados y tenía algunos ahorrillos.

Pero para mi hermano Nacho fue un desastre. Él terminó después de dos años de desarrollo porque le cambiaron todo lo que había hecho al principio (al principio el personaje era mucho más pequeño y en una misma pantalla había varias alturas, no como en el definitivo que había como mucho dos) y cuando en aquel entonces podías tardar menos de un año en hacer un juego, allí se terminó en dos. Dos años con cero ingresos, desesperado porque lo quería acabar y no paraban de cambiarle cosas y cuando lo termina y lo sacan casi no le pagaron nada, a pesar de que el juego había vendido bien.

Cuando en 1999 creamos FX Interactive y Nacho fue la primera persona que contratamos, me dijo Pablo Ruiz que para él seguía siendo uno de sus máximos acreedores por todo lo que no le pagó en aquella época (aunque luego no pareció acordarse mucho de aquello cuando mi hermano decidió dejar FX para venirse a crear Crocodile conmigo). Todo esto hizo que mi hermano dejase los juegos, dejase de encerrarse conmigo en la habitación del garaje de casa de mis padres con el ordenador, perdiese la ilusión, y no quisiese saber nada cuando empezamos a hacer cosas y creamos Dinamic Multimedia.

**En FX Interactive se toma la decisión de hacer un remake de Navy Moves y de contar con el programador original de 8 bits en su desarrollo, ¿cómo surge esa idea y cómo la recibes tú, Ignacio?**

*C*: En FX Interactive no teníamos presupuesto para hacer dos desarrollos al inicio. Así que como yo venía de hacer muchos, muchos PCFútbols y derivados y me venía bien un descanso pero Víctor, al contrario, tenía ganas de hacer un desarrollo con equipo propio, se decidió que Víctor empezaba haciendo un desarrollo que duraría un año y medio y después con el dinero que diese ya se podría intentar ir creando un segundo equipo con el que yo haría un juego.

Además, así yo podría hacer I+D para la compañía ya que yo siempre había sido el miembro más técnico y en los últimos tres PCFútbols me había dedicado a la producción pura y no había programado prácticamente nada. Luego la empresa se centró, prácticamente en exclusiva, en la parte de edición de productos externos y el desarrollo quedó en un segundo plano.

Cuando creamos FX Interactive, a Nacho ya le había vuelto la ilusión por desarrollar juegos y eso hizo que pidiese formar parte del equipo y como Víctor guardaba muy buen recuerdo… Fue la primera persona que contratamos en FX.

La idea de que fuese un Navy Moves no creo que fuera del momento. En aquel entonces Víctor se había comprado su primera consola, una PS2 y tenía el juego Metal Gear con el que estaba encantado. Así que él lo que quería hacer era un juego de intriga y acción bélica lo que rápidamente le llevó a sus primeros títulos.

*I*: En ese momento yo estaba deseando irme de la empresa en la que estaba y Víctor me llamó para colaborar con él en el desarrollo de su idea. A partir de ahí, comenzamos a trabajar en el denominado, por aquel entonces, "Proyecto 941".

Como llevaba mucho tiempo sin hacer nada relativo con los juegos, no sabía nada de lo que significaba DirectX ni similar, ni cómo usarlo; por eso cada nuevo paso que daba durante los primeros meses lo disfrutaba al máximo.

Para mí fue el momento de volver a un tipo de desarrollo que me gustaba mucho. Siempre me ha gustado mucho programar y hacer cosas nuevas y en la industria del videojuego es de los pocos sitios donde, a nivel de programación, siempre estás evolucionando.







*Zack Zero: El futuro cercano de los hermanos Abril (Imágenes exclusivas)*

**Actualmente os encontráis desarrollando un nuevo juego, Zack Zero, en una empresa que ambos habéis formado junto a otro colega de Dinamic Multimedia. ¿qué nos podéis contar sobre este proyecto? Comentáis que tiene varias referencias a Phantis...**

*C*: Mi idea inicial era hacer un juego muy sencillo, de plataformas 2D con gráficos en 3D pero muy sencillote, para poder desarrollarlo en año y medio y hacer pruebas de mercado sin que el principal objetivo fuese, ni mucho menos, el beneficio económico. Como en aquel entonces estábamos en el 20 aniversarío de Phantis y su segunda carga era un concepto de juego sencillo pero variado y con una estructura de mapa ya definida, decidimos que un remake podría estar bien: gran parte del juego estaba ya definido y sólo teníamos que centrarnos en crear la tecnología, crear los gráficos, definir el nuevo comportamiento de los enemigos (en Phantis eran demasiado simples) y 'poco' más.

Sin embargo, cuando Alberto Moreno, nuestro socio, y yo fuimos detallando y evolucionando el diseño, nos fueron surgiendo ideas que nos entusiasmaban pero que no queríamos meter en este juego para no alargar su desarrollo.

Pero como la creación de la propia empresa se complicó y la situación personal de alguno de nosotros cambió, pues decidimos disfrutar un poco más de lo que estábamos haciendo, aunque nos llevara más tiempo y aunque no fuese el concepto ni la idea inicial. Según esto, aunque olvidamos la idea de remake de Phantis, mantuvimos más o menos la estructura de niveles: superficie, base, gruta, etc. aunque cada una la complicamos bastante más. Y luego hemos dejado alguna otra referencia a Phantis.

En la web actual (http://www.crocoware.net) hay alguna pantalla y una pequeña descripción de hace casi un año. Esta web la hicimos de cara a nuestro primer GDC como Crocodile y para que a la gente con la que

contactáramos viera que existía algo. Ese fue nuestro objetivo y por eso no nos hemos preocupado en actualizarla. En uno o dos meses la cambiaremos de cara a empezar a enseñar un poco el juego.

Josetxu Malanda













# TIRA CÓMICA

## ANÁLISIS










## CURSO DE CÓDIGO MÁQUINA







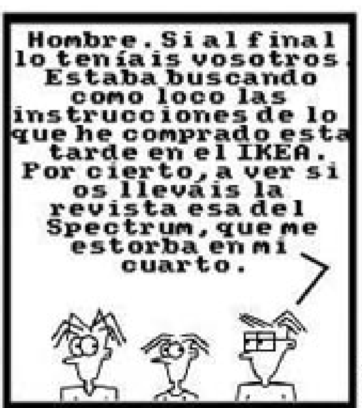


## HARDWARE









**VERSIÓN IMPRESA**













Pedrete



# NO PASA NADA

Pueden llamarme chulo, pero evitaré caer en la vulgar tropelía de recurrir al primer ejemplar de la presente publicación para revisar las declaraciones de su editorial. ¿Me permiten, pues, un levemente arriesgado ejercicio adivinatorio al respecto?

Anoten, a modo mental, algunas de las expresiones y giros lingüísticos que allí se debieron de incluir con sospechable seguridad: "grupo de aficionados", "humilde aportación a la escena", "ilusión con la que nace este proyecto"… parafraseos todos ellos ya conocidos por el apreciado lector, pues no ha lugar a dudar que también vdes. mismos han recurrido a ellos para dar merecido cojín protocolario de salida a sus respectivas iniciativas. Evitemos el engaño: probablemente los cabezas visibles de MagazineZX tienen tanto de excelso y privilegiado en materia gris como lo que usted o yo de criadores de canarios-flauta: bípedos ellos, machacas del bolígrafo o teclado para con sus sufridos artículos, juntamnemónicos ocasionales, otros antes bien ávidos de mostrar sus reflexiones en negrita. Buscando y a su vez ofreciendo, en definitiva, ese "espíritu del cinco de noviembre" (si los lóbulos de don Arias Navarro me permiten la licencia) que por esos entonces aún emanaba la misa perpetua por Nuestra Señora de MicroHobby, con el latente deseo de que la pueril letra de cada uno de ellos fuera en cualquier momento revisada, corregida, vilipendiada o hasta enculada por terceros antes que acribillada por racimos de inocuos alabos de tampón. Nada que no responda a una modesta y primigenia ambición que, como ya comentaba, también hemos experimentado tanto usted como un servidor.




Pecaríamos de hombres desactualizados si no admitiéramos que, pasados seis años desde su alumbramiento, el contexto bajo el que la revista vio la luz ha cambiado considerablemente. "¡Celebremos estar vivos!", proclamó el honorable doctor Barnard en uno de sus más afamados discursos… y poco se podía imaginar dicho cardiólogo que su exhortación sería tan cuantitativamente aplicable, años después, por lo que a la aldea global se refiere: aquí estamos cada uno de nosotros echando mano de twitters, facebooks, pdfs caseros y demás flora para reclamar, ¡qué menos!, un mínimo espacio y éter para nuestros gritos binarios.

Y metidos en generalizaciones, es perfectamente normal que, con menos o más brevura, todos hayamos sucumbido al mismo virus. Ya ve usted; ahora va a resultar que ésta nuestra comunidad virtual ya tenía más que interiorizado el ínclito 'Yes, we can' mucho antes de que éste empezara a oler a chamusquina: el 'yo también existo' como santo y seña; el llano y consecuente camino a la aceptación en general y a la (casi siempre) postrera ansia de autoaceptación en particular…

Meridiana resulta la comprensión no del todo adecuada de tal retahíla de herramientas. Ahí están nuestras nuevas publicaciones, sometidas a la frágil continuidad que sólo a posteriori hemos sabido detectar. Ahí quedan las entradas a nuestros portales y blogs; paridos y criados con vocación de Kaaba de ese vintage que nos concierne, y que saciamos con únicas e intransferibles sapiencias, pero que a plazo indefinido acaban por convertirse en perfectos ventiladores de vanidades, complejos, amagos autolesivos y hasta rencillas de trinchera. Blogs sin receta previa y de posología a la carta, a los que sin ápice de duda hoy día se referiría Warhol en su imperecedera profecía, dejando definitivamente de lado a la caja tonta como símil de referencia. Maestro Onán; cuánto darías por reclamar un merecido descanso a tu memoria…

Somos, si se me permite, víctimas de una saciedad mal digerida. Y no sólo hemos aprendido a convivir con el cólico, sino que ya no nos vemos correspondidos si no lo embutimos y proferimos por él nuestra profunda estima. ¿O me va a negar, acaso, que de entre sus opulentas selecciones de emepetreses de tapadillo se habrá escuchado apenas un diez por ciento de los mismos? Aplique rasero similar para cuantos cientos de .tzx se le ocurra imaginar; para cuantas enésimas revisiones de Dustin o Target Renegade haya parido madre o estén en camino de engendrarse; para cuantos proyectos hardware o brillantes nuevas rutinas hayan sido o sean, respectivamente, implementados o mejoradas. Razone muy, pero que muy desapasionadamente si es coherente pensar que somos capaces de provocar las mismas erecciones ajenas de antaño… y más aún, piense hasta qué punto y con qué frecuencia esas loables labores ajenas se la ponen dura a usted. Nadie, llegada tal saturación de outputs potenciales, es capaz de juzgar correctamente su calentura a estas alturas sin empezar a confundir preocupantemente Naomis Watts con ocasionales Supremas de Móstoles al uso. Pero

no se alarme en mesura alguna. Nada mórbido (le) acontece; compartimos vivencias al respecto (?).

Y con tantas tintas cargadas de por medio a base de fanzines, entradas blogueras y demás, hemos ido olvidando paradójicamente al gran calamar. Ése al cual apuntamos todos y al que, aun pretendiendo alimentarlo, puede que estemos subministrando complejos vitamínicos tan generosos como a la par estériles (si no han tenido a bien acceder a la fábula de Arturo y Clementina de Adela Turín, háganse con una copia de la misma y me agradecerán el favor, al tiempo que se situarán en las coordenadas precisas). Un calamar al que comúnmente venimos conociendo como 'escena', y que si de algo adolece es que en su jodidamente ya larga trayectoria todavía no hemos sido capaces de poderle hacer una instantánea; que esperamos, para tal fin, que en algún momento se le ocurra sonreír, cuando lo que en realidad está deseando es que se le encuentren un rostro y una entidad finalmente tangibles; un calamar al cual tanto omega-3 entusiástico, tantísimo software de nueva factura, tanto review y tanto guarismo en participación forera le ha conllevado una atrofia aguda de movimientos peristálticos por no ofrecérsele tal banquete mínimamente masticado tal y como merece. Que sabe que la suma de tantas partes no forman su todo, y por tanto, que por mucho que ilusionadamente saquemos la regla para medir (y especialmente medirnos, mutua y recíprocamente) nuestras respectivas chorras, no conseguiremos sacar al único y gran falo escénico de su preocupante flacidez.

Como ente integrante de la movida (por así decir), MagazineZX no habrá posiblemente escapado de esa inercia. Decía hará unos tiempos, y literalmente, un tal Alfonso Armada, en su retiro gallego y a raíz de su golpe de timón interruptus: "en mi juventud yo devoraba los periódicos, veía la televisión, estaba constantemente pendiente de cuanto en el mundo ocurría o dejaba de ocurrir… hasta que por fin me di cuenta de que ese mundo seguía tranquilamente su curso, sin que yo debiera de hacer nada por influir en ello". Al campechano ex-general, ya en obligada reserva, es obvio que le hubieran convenido unos efectivos pokes a tiempo. Aliviémonos por ello, pero a su vez quedémonos con el poso de su reflexión: a nosotros, que sí disponemos de ellos, se nos han atragantado cosa mala las vidas infinitas. Nos hemos convertido, fíjense, en brillantes músicos; músicos, como Mario Abraham Kortzclap, especializados en la versátil tonalidad de Mi Mayor. Porque la carne es débil; pero quién sabe si también esa flaqueza haya acabado salpicando todo lo que afecta (sin orden ni prioridad expresa) a esos 'mi' artículo, 'mi' fanzine, 'mi' programa, 'mi' avatar, 'mi' dominio blogspot o, en certero destino, a "mi opía" (Luis Sánchez Pollack dixit). Voluntario me ofrezco si es preciso, quede dicho, a empezar la cola de aquellos que, como el menda, interpretan (y creo que interpretan con razón) que de humanos es errar.

Sobra rasgar vestidura alguna, en conclusión. Si me permiten recuperar el referenciario ubicado en Armada y la llamada transición, les presto una segunda mención literal de otro de los personajes paradigmáticos de dicho periodo histórico; en este caso, del otrora cabal Adolfo Suárez, quien todavía en tiempos del régimen tuvo la nada deleznable osadía de sugerir: "vamos a elevar a la categoría de normal lo que a nivel de calle es simplemente normal".

Amén Jesús, señor duque. Los propósitos iniciáticos de MagazineZX puede que hayan cumplido su comedido, o bien es juzgable que no haya sido así. Los popes culpables de la misma, en cualquier caso, consideran cerrado el ciclo que le ha tocado en suerte vivir. Obviemos coletillas y desquites al estilo "cierre definitivo", "no hay vuelta atrás", "caiga quien caiga"…; saben estos señores, desde la más absoluta serenidad, que ni va a caer ni debe caer nadie. Todo forma parte de la normalidad, de esa normalidad de la que hoy apetecemos beber.

A otra cosa, pues. Y valga esto como ejemplo fiel de lo que va a implicar la acción futura de sus integrantes (a decir verdad, también lo ha sido y lo fue presente y pretérita). Siguen y seguirán sus actividades individuales y conjuntas de pestaña quemada desde el mismo instante en que cesa ésta su iniciativa menos silenciosa. Boicot al decibelio que no debiera resultar extraño para el lector que sepa y conozca de ellos. Se abre un poquito más de tiempo, para ellos digo, de cara a optimizar sus respectivas actividades de la vida diaria; de cara al cultivo interpersonal. Por supuesto, también en pos de la cultura escénica de carne y hueso. Y más aún, de cara a seguir sintiendo en ocho bits sin que por ello tal base sea el patrón neuronal al cual verse encorsetado.

Mando, a modo de albricias y congratulación, generoso ramo de camelias a sus responsables; camelias encargadas directamente desde el Pazo de Santa Cruz de Ribadulla (al parecer, unas de las más selectas de la provincia, de tan aserrado contorno como perenne follaje). A ellos, transmitida quede la tranquilidad añadida de que la escena seguirá incólume pese a su decisión. Y que no pasa (ni pasará) nada por tal circunstancia, sea o no en consecuencia directa a su alto en el camino.

Una vez más: celebremos estar vivos, doctor Barnard.

Albert Valls

